Ein Vielkernprozessor mit Laufzeitüberwachung zur Separierung unterschiedlich kritischer Anwendungen

A Many-Core Platform with Run-Time Monitoring to Support Separation of Mixed-Criticality Applications

Von der Carl-Friedrich-Gauß-Fakultät
der Technischen Universität Carolo-Wilhelmina zu Braunschweig

zur Erlangung des Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

genehmigte Dissertation

von
Boris Alexander Motruk
geboren am 21.08.1980
in Salzgitter

Eingereicht am:	17.03.2014
Disputation am:	28.07.2014
1. Referent:	Prof. Dr.-Ing. Mladen Berekovic
2. Referent:	Prof. Dr.-Ing. Harald Michalik

2014

Kurzfassung

Mehr- und Vielkernplattformen bieten ausreichend Ressourcen für eine weitere Steigerung der Rechenleistung, zum einen für aufwendigere Anwendungen und zum anderen für die Integration mehrerer Anwendungen, welche sonst auf mehreren separaten Plattformen ausgeführt würden. Die große Anzahl an Ressourcen einer Vielkernplattform kann weiterhin dafür verwendet werden, einer Anwendung mehr Ressourcen als nötig redundant zuzuweisen oder zunächst unbenutzte Komponenten dazu zu verwenden, fehlerhafte Komponenten zur Laufzeit zu ersetzen, um so die Zuverlässigkeit und Verfügbarkeit von Anwendungen zu erhöhen. Hierfür muss eine Vielkernplattform eine transparente und flexible Zuordnung von Anwendungen erlauben, welche sich auch zur Laufzeit ändern lässt. Dasselbe gilt für Kommunikationsverbindungen zwischen Anwendungen auf unterschiedlichen Prozessoren sowie zwischen Anwendungen und verteilten Peripheriemodulen oder Speicherschnittstellen.

Die vorliegende Arbeit präsentiert eine parametrisierbare und synthetisierbare Vielkernplattform, welche die genannten Bedingungen durch Virtualisierung sämtlicher Ressourcen erfüllt. Die Plattform dient zur Erforschung neuer Mechanismen zur Separierung unterschiedlich kritischer Anwendungen. Sollen nämlich unterschiedlich kritische Anwendung ohne eine ausreichende Separierung gemeinsam auf einer Vielkernplattformen integriert werden, müssen alle Anwendungen die Anforderungen der Anwendung mit der höchsten Kritikalität erfüllen. Dies würde vor allem den Aufwand für weniger kritische Anwendungen stark erhöhen. Eine ausreichende Separierung ermöglicht die unabhängige Entwicklung und Zertifizierung einzelner Anwendungen sowie deren kostengünstige Neuzertifizierung nach einer eventuellen Aktualisierung.

Die Separierung betrifft nicht nur die Unabhängigkeit der einzelnen Anwendungen in Bezug auf ihr Zeitverhalten und ihren Raumbedarf, sondern muss auch auf ihren Energieverbrauch erweitert werden, da die verfügbare Energie ebenfalls von allen Anwendungen gemeinsam genutzt wird. Ein erhöhter Energieverbrauch einer einzelnen Anwendung kann die verfügbare Energie für andere Anwendungen einschränken und durch eine erhöhte thermische Belastung die Verfügbarkeit und Lebensdauer des gesamten Chips reduzieren.

Neben der statischen Separierung unterschiedlich kritischer Anwendungen durch eine exklusive Zuweisung von Ressourcen bietet die im Rahmen dieser Arbeit vorgestellte Plattform eine skalierbare Laufzeitüberwachung mit einer kurzen Reaktionszeit, welche eine sichere und effiziente gemeinsame Nutzung von Ressourcen erlaubt. Die entwickelte Laufzeitüberwachung ermöglicht die Überwachung des spezifizierten Verhaltens der einzelnen Anwendungen und kann dieses bei Bedarf zur Laufzeit erzwingen. Insgesamt ist die Arbeit ein weiterer Schritt, um die Vorteile von Vielkernplattformen für sicherheitskritische und unterschiedlich kritische Anwendungen effizient nutzbar zu machen.

Abstract

Modern multi- and many-core platforms offer sufficient resources for further increasing the performance of advanced applications. Moreover they allow integrating multiple applications that formerly ran on multiple chips. The large amount of resources can additionally be used to map applications redundantly to more resources than required to increase reliability. Spare parts can be used to replace faulty components at run time for higher availability. A suitable platform must allow remapping of applications and replacement of peripherals dynamically. Mapping to distributed resources but also communication among resources ideally is transparent and flexible to allow changes at run time.

In this thesis, a parameterizable and synthesizable many-core platform is presented, which realizes the requirements above by virtualizing all resources that are available on the platform. The platform is used as a research vehicle to develop mechanisms for separating applications of different criticalities on a shared platform.

On a many-core platform that runs mixed-criticality applications, all applications have to be sufficiently separated. Otherwise all applications have to fulfill the requirements of the highest level of criticality, even low critical ones. This would significantly increase the costs of a shared platform. Separation enables individual development and certification of applications and cost-efficient re-certification of single applications after an update.

Separation does not only include independence in terms of time and space, but also in terms of power consumption as the available energy for a many-core system is shared between all running applications. Increased power consumption of one application may reduce the available energy for other applications or the reliability and lifetime of the complete chip.

Beside static separation of mixed-criticality applications by assigning them to separate resources, a fast and scalable monitoring and control mechanism allows safe and efficient sharing of resources by enforcing specified behavior of applications at run time. All in all, this thesis' contribution is a step towards exploiting the benefits of multi- and many-core platforms for mixed-criticality applications.

Danksagung

Hiermit möchte ich meinem Betreuer Professor Dr.-Ing. Mladen Berekovic für die Möglichkeit und das Vertrauen danken, mit vielen Freiheiten an einem sehr interessanten Thema arbeiten zu können. Auch Professor Dr.-Ing. Harald Michalik möchte ich für die Übernahme des zweiten Gutachtens besonders danken.

Weiterhin danke ich den Mitarbeitern der Abteilung Entwurf integrierter Systeme sowie Professor Dr.-Ing. Rolf Ernst und den Mitarbeitern seiner Arbeitsgruppe Architektur und Entwurf eingebetteter Systeme für zahlreiche interessante Gespräche, Inspirationen und Rückmeldungen zu meiner Arbeit.

Besonderer Dank gilt hierbei Dr. Rainer Buchty für die Anleitung zum wissenschaftlichen Arbeiten und der konstruktiven, ausführlichen und zeitnahen Besprechung der vorliegenden Arbeit und vorangegangener Veröffentlichungen. Weiterer besonderer Dank geht an Jonas Diemer für die Anpassung des in dieser Arbeit verwendeten NoCs und zahlreiche wertvolle Diskussionen.

Weiterhin möchte ich den studentischen Mitarbeitern Arthur Martens und Jan Wagner für ihren Beitrag zu der Interrupt-Übersetzung beziehungsweise der DDR2-Steuerung der IDAMC-Plattform danken.

Mein besonderer persönlicher Dank geht an meine Eltern Cornelia und Heinz, für die Möglichkeit, eine gute Ausbildung absolvieren zu können, für den beständigen Rückhalt, die vielfältige Unterstützung sowie für die vielen Freiheiten und das Vertrauen, das sie mir entgegengebracht haben. Nicht zuletzt geht großer Dank an meine Ehefrau Marion und unsere gemeinsamen Kinder Felix, Hannah und Neele, dafür dass sie meinen Blick stets immer wieder auf die wirklich wichtigen Dinge des Lebens gelenkt haben.

Inhaltsverzeichnis

1 Einleitung

Durch die Erhöhung von Frequenz oder Komplexität kann die Leistung von Prozessoren nicht mehr weiter erhöht werden, jedenfalls nicht unter einem vertretbaren Energieverbrauch[21]. Stattdessen werden immer mehr, aber dafür einfachere Rechenkerne auf einem gemeinsamen Chip integriert. Bereits zu der Zeit der Verfassung dieser Arbeit sind die meisten produzierten Prozessoren Mehrkernsysteme. Die Anzahl an möglichen Rechenkernen ist jedoch, bei Verwendung eines gemeinsamen Busses für die Verbindungen zwischen den Elementen auf dem Chip, durch dessen Bandbreite und Verzögerung limitiert. Für Plattformen mit einer größeren Anzahl an Rechenkernen kommen daher skalierbare Verbindungen wie *Network-on-Chips (NoCs)* zum Einsatz. Auch hier existieren bereits einige Forschungsplattformen, wie beispielsweise Intels SCC [11], sowie die ersten kommerziellen Systeme, wie zum Beispiel Tileras TILE64-Prozessor [14].

Die verfügbare hohe Rechenleistung solcher Vielkernprozessoren wird zum einen für immer komplexere Anwendungen verwendet. Sie kann aber auch verwendet werden, mehrere Anwendungen, die vorher auf separaten Systemen liefen, auf einer gemeinsamen Plattform zu integrieren, um so vor allem Kosten zu sparen. Des Weiteren bieten Vielkernprozessoren ausreichend Ressourcen, damit Anwendungen mehr Ressourcen als nötig zugeordnet werden können, um so deren Verfügbarkeit und Zuverlässigkeit zu erhöhen. Die zusätzlichen Ressourcen könnten dann verwendet werden, um fehlerhafte Teile zur Laufzeit zu ersetzen. Aus denselben Gründen sind Vielkernprozessoren auch für sicherheitskritische und eingebettete Systeme attraktiv. Die meisten Mikrocontroller-Anbieter haben Mehrkernprozessoren bereits im Angebot. Aktuell befinden sich vorwiegend Zwei- oder Vierkernmodelle im Einsatz, es sind jedoch auch bereits Achtkernmodelle verfügbar.

Eingebettete Systeme müssen zum einen mehr und mehr Funktionalitäten in unterschiedlichen Einsatzbereichen unterstützen, wie beispielsweise im Automobil oder der Luftfahrt, und zum anderen immer aufwendigere Funktionen ermöglichen, wie zum Beispiel Fahrerassistenzsysteme (engl. *Advanced Driver Assistance Systems*). Gerade im Automobil, wo durch die gestiegene Nachfrage nach immer mehr Unterhaltungs-, Kommunikations-, Komfort- und Sicherheitsfunktionen bis zu über hundert meist vernetzte Steuergeräte zum Einsatz kommen, bietet die Integration mehrerer Funktionen auf einer gemeinsamen Plattform ein großes Kosteneinsparungspotential, schon allein durch die dann geringere Anzahl an benötigten Steuergeräten. Weiterhin wird weniger Platz benötigt, um die geringere Anzahl, und damit auch weniger Verbindungen zwischen den Steuergeräten, im Fahrzeug unterzubringen. Verbindungen zwischen Rechenkernen auf demselben Chip sind darüber hinaus schneller und weniger fehleranfällig als Verbindungen zwischen Prozessoren in separaten Steuergeräten. Neben den Kosten und dem Volumen lässt sich über die Reduzierung der Anzahl an Steuergeräten außerdem noch das Gewicht und auch der Energieverbrauch der Elektronik deutlich verringern.

Um die zusätzlichen Ressourcen einer Vielkernplattform zur Erhöhung der Fehlertoleranz einzusetzen, ist es erforderlich, dass sich Anwendungen flexibel den verfügbaren Ressourcen zuordnen lassen und sich diese Zuordnung auch zur Laufzeit ändern lässt. Um die Komplexität der Anwendungen

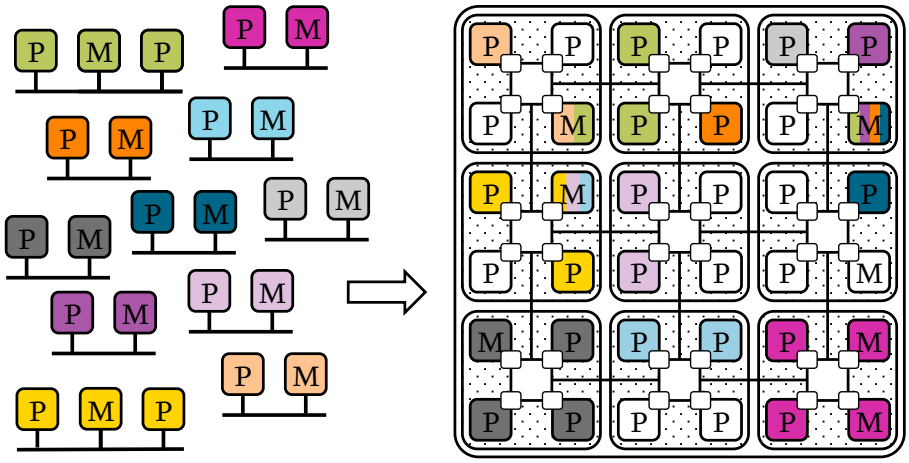

Abbildung 1.1.: Integration mehrerer Anwendungen auf gemeinsamer Plattform

durch die benötigte Flexibilität nicht zu erhöhen und besonders auch, um bereits existierende Anwendungen ohne Änderungen auf eine Vielkernplattform portieren zu können, sollte die Zuordnung daher für die Anwendung selbst transparent geschehen. Anwendungen benötigen dadurch keinerlei Kenntnisse über die physischen Rechenkerne, Speicherschnittstellen und Peripheriemodule und deren Lage auf dem Chip. Diese Transparenz- und Flexibilitätsanforderungen gelten nicht nur für die Zuordnung von Anwendungen, sondern auch für deren Kommunikation mit angeschlossenen Komponenten und untereinander. Sollte eine Anwendung zur Laufzeit auf einen anderen Prozessor migriert werden, müssen gleichzeitig auch alle zugehörigen Kommunikationsverbindungen für die Anwendung selbst transparent angepasst werden.

Abbildung 1.1 zeigt auf der linken Seite mehrere Anwendungen, die zunächst auf separaten herkömmlichen Einzel- und Mehrprozessorsystemen laufen. Diese sind hier als Prozessoren P und Speicher M, die über einen Bus verbunden sind, symbolisiert. In Abbildung 1.1 rechts werden diese Anwendungen auf einer gemeinsamen Vielkernplattform integriert. Hierbei werden manche redundant ausgelegt, wie die dunkelgrau dargestellte Anwendung, um deren Verfügbarkeit und Zuverlässigkeit zu erhöhen. Das kann je nach Bedarf nur einzelne Teile, wie beispielsweise den Prozessor oder aber auch sämtliche Komponenten betreffen, wie zum Beispiel bei der in pink dargestellten Anwendung. Andere Teile, wie zum Beispiel externe Speicherschnittstellen werden, um die Effizienz zu erhöhen oder auch aus Mangel an Alternativen, von mehreren Anwendungen geteilt. Weiterhin werden Ersatzressourcen vorgehalten, die zunächst keiner Anwendung zugeordnet werden, um defekte Komponenten zur Laufzeit ersetzen zu können.

Wenn unterschiedlich kritische Anwendungen gemeinsam auf einer Plattform integriert werden sollen (engl. *Mixed-Criticality*), wie beispielsweise Unterhaltungsfunktionen und eine elektronische Stabilitätskontrolle, ist die Situation nicht mehr ganz so einfach. Es existieren Normen, die Anforderungen an kritische Anwendungen definieren, um deren funktionale Sicherheit zu gewährleisten. Dies wären beispielsweise die IEC 61508 [51], eine Norm für elektrische, elektronische und programmierbar elektronische Systeme im Allgemeinen oder, davon abgeleitet, die ISO 26262 [52], eine Norm speziell für Kraftfahrzeuge. Funktionale Sicherheit ist definiert als die Freiheit von inakzeptablen Risiken. Was inakzeptable in dem Kontext bedeutet, definieren ebenfalls die Normen.

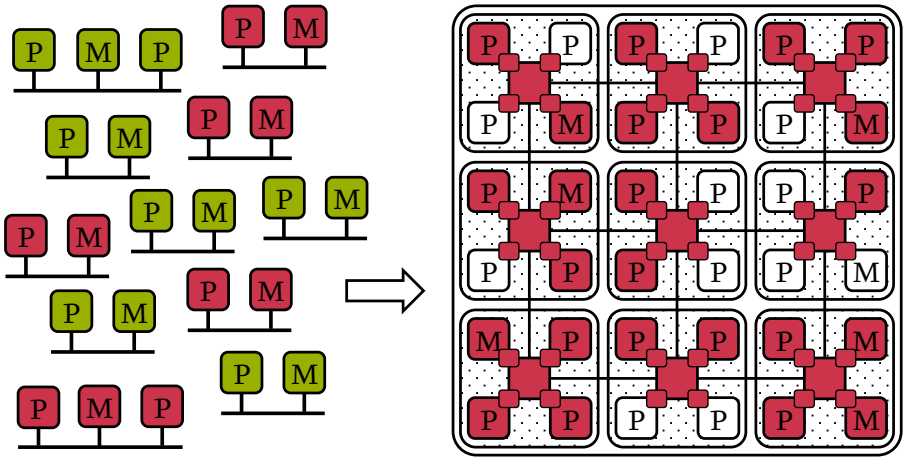

Abbildung 1.2.: Integration ohne ausreichende Separierung

In den Normen werden Anforderungen für alle Phasen eines Produktlebenszyklus, wie Entwurf, Implementierung, Produktion, Test, Wartung und Entsorgung beschrieben, um das Restrisiko für gefährliche Ereignisse zu reduzieren. Hierbei werden vier diskrete Stufen eingeführt, welche in der IEC 61508 *Sicherheitsintegritätslevel (SIL)* und in der ISO 26262 *Automotive-Sicherheitsintegritätslevel (ASIL)* genannt werden. Umso höher die Stufe, desto mehr und strengere Anforderungen müssen erfüllt werden, was mit einem höheren Entwicklungs- und Zertifizierungsaufwand einhergeht.

Wenn nun unterschiedlich kritische Anwendungen auf einer Plattform integriert werden sollen, fordern die Normen eine ausreichende Trennung, damit Ausfälle von nicht oder weniger kritischen Anwendung nicht zu Fehlern und Ausfällen bei höher kritischen Anwendungen führen können. Wenn eine solche Trennung nicht nachgewiesen werden kann, müssen auch weniger kritische Anwendungen dieselben Anforderungen erfüllen wie hoch kritische Anwendungen, was zu stark ansteigenden Kosten des Gesamtsystems führen würde. Darüber hinaus sind manche weniger kritische Anwendungen, beispielsweise aus dem Multimediabereich, so komplex, dass diese sich gar nicht oder jedenfalls nicht mit einem vertretbaren Aufwand entsprechend einem hohen Kritikalitätslevel analysieren beziehungsweise zertifizieren lassen würden.

Abbildung 1.2 links zeigt ein Beispiel mit kritischen Anwendungen in rot und nicht kritischen Anwendungen in grün, welche zunächst auf separaten Plattformen laufen und so physisch voneinander getrennt sind. Werden diese nun, wie in Abbildung 1.2 rechts, auf einer gemeinsamen Vielkernplattform ohne ausreichende Separierung integriert, werden alle Anwendungen kritisch, da eine gegenseitige Beeinflussung nicht ausgeschlossen werden kann. Durch die hiermit erhöhten Anforderungen für nicht oder weniger kritische Anwendungen würde der Aufwand hierbei nicht nur für die Integration steigen, sondern würde auch jedes Mal wieder anfallen, wenn eine Anwendung aktualisiert wird, was ohne ausreichende Separierung zu einer Neuzertifizierung, nicht nur der aktualisierten Anwendung, sondern sämtlicher Anwendungen auf der geteilten Plattform führen würde.

Eine mögliche Umsetzung der geforderten Separierung könnte darin bestehen, jegliches Teilen von Rechenkernen, Speichern, Peripheriemodulen, Kommunikationsverbindungen und der Energieversorgung möglichst zu unterbinden. Dies würde jedoch die Vorteile einer Vielkernplattform

stark einschränken. Alternativ könnte durch eine exakte Ablaufplanung aller Aufgaben des Systems, unter Berücksichtigung gegenseitiger Beeinflussungen, sichergestellt werden, dass kritische Anwendungen nur so weit von anderen Anwendungen beeinträchtigt werden können, dass auch im schlechtesten Fall die erfolgreiche Ausführung garantiert werden kann. Hierfür benötigt man, neben der Ablaufplanung, Ereignismodelle von allen Aufgaben, die auf der Plattform gemeinsam ausgeführt werden sollen, sowie den Ressourcenverbrauch der einzelnen Aufgaben pro Aktivierung in Zeit- oder Energieeinheiten. Die Ereignismodelle beschreiben dabei die maximale und minimale Anzahl an Aktivierungen pro Zeitintervall beziehungsweise den minimalen und maximalen Abstand zwischen aufeinanderfolgenden Aktivierungen. Bei der Analyse des Systems interessiert vor allem die Maximalanzahl beziehungsweise der Minimalabstand, da das Verhalten im schlechtesten Fall für die sichere Separierung wichtig ist.

In der Analyse der einzelnen Aufgaben müssen alle Möglichkeiten der Beeinflussung durch andere Aufgaben betrachten werden, wie beispielsweise gemeinsam verwendete Ressourcen und eingehende Kommunikation. Die Zugriffe auf gemeinsam benutzte Ressourcen, wie zum Beispiel Speicherschnittstellen, durch andere Aufgaben geht dann als zusätzliche Verzögerung in die Analyse ein. Die zusätzliche Verzögerung wird über die Anzahl aller Zugriffe von anderen Aufgaben in einem beliebigen Zeitfenster und die dafür benötigte Zeit bestimmt. Eingehende Nachrichten sind weitere aktivierende Ereignisse, die im Ereignismodell der aktivierten Aufgabe beinhaltet sein müssen.

Bei der Analyse einer kritischen Aufgabe müssen die Ereignismodelle aller relevanten Aufgaben, welche die untersuchte kritische Aufgabe beeinflussen könnten, als Eingangsparameter der Analyse denselben hohen Anforderungen genügen. Sämtliche Aufgaben, die also beispielsweise denselben Speicher benutzen oder mit der kritischen Aufgabe kommunizieren, müssen damit den Anforderungen desselben Sicherheitsintegritätslevels wie die analysierte kritische Aufgabe entsprechen. Hierdurch würden, wie bereits angemerkt, die Kosten für weniger kritische Anwendungen stark ansteigen, wenn sie gemeinsam mit hoch kritischen Anwendungen auf einer Plattform integriert würden. Alternativ könnte sämtliches Fehlverhalten von weniger kritischen Aufgaben, also Abweichungen vom Ereignismodell, in der Analyse kritischer Aufgaben betrachtet werden. Hierdurch würde sich das Analyseergebnis jedoch deutlich vom tatsächlichen Verhalten unterscheiden, was zu einer starken Überdimensionierung beziehungsweise Unterauslastung des Systems führen würde.

In dem in der vorliegenden Arbeit verfolgten Ansatz wird das Verhalten von weniger kritischen Aufgaben zur Laufzeit überwacht und gegebenenfalls das Verhalten, welches während der Analyse angenommen wurde, erzwungen. Die Überwachung ist in den Verbindungspunkten zwischen dem Netzwerk der Vielkernplattform und den damit verbundenen Elementen implementiert, um so fehlerkonservierende Teilsysteme zu erzeugen. Wenn unterschiedlich kritische Anwendungen nun auf Prozessoren in unterschiedlichen Teilsystemen implementiert werden, können mögliche Fehler in unterschiedlichen Anwendungen unabhängig gemacht werden. Jedes Element kann so ausfallen, ohne andere Elemente oder das Netzwerk zu beeinflussen. Dies erfordert, dass die in dieser Arbeit vorgestellten Netzwerkschnittstellen, die Anforderungen der Aufgabe mit der höchsten auf der Plattform vorkommenden Kritikalität erfüllen müssen. Durch diesen einmal zu erbringenden Aufwand können jedoch die Anforderungen von weniger oder nicht kritischen Anwendungen auf dem Level belassen werden, welches nötig wäre, wenn diese Anwendungen exklusiv auf einer Plattform ausgeführt würden. Abbildung 1.3 rechts verdeutlicht dies. Die Netzwerkschnittstellen werden als kritisch eingestuft, die weniger kritischen Anwendungen werden weiterhin in grün

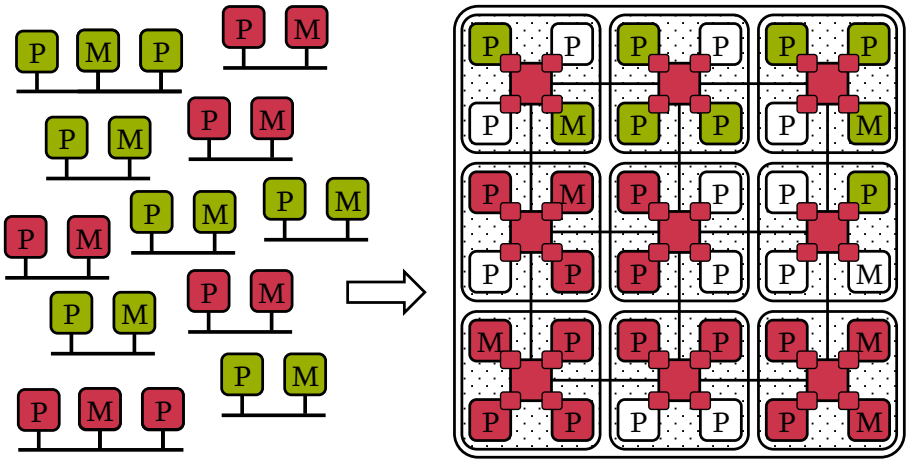

Abbildung 1.3.: Integration mit ausreichender Separierung

gezeichnet, die an sie gestellten Anforderungen erhöhen sich in diesem Fall nicht und bleiben auf dem niedrigen Level wie in Abbildung 1.3 links dargestellt.

Die in den Normen zur funktionalen Sicherheit geforderte Partitionierung von Zeit und Raum ist für die Fehlerkonservierung in einem sicherheitskritischen System zwingend notwendig [48]. Darüber hinaus ist Energie eine weitere Ressource, die auf einer gemeinsam genutzten Plattform bei übermäßigem Verbrauch durch eine Anwendung, andere Anwendungen negativ beeinflussen kann. Daher muss die geforderte Separierung von Rechenzeiten, Netzwerkverzögerungen, Bandbreiten und Speicherplatz auf die Separierung des Energieverbrauchs erweitert werden. Das insgesamt zur Verfügung stehende Energiebudget wird zur Entwurfszeit abhängig von einer eventuell vorhandenen Lüftung, dem Gehäuse, der Batteriekapazität, falls zutreffend, und den Umgebungsbedingungen festgelegt. Dieses Budget wird zur Laufzeit von allen Anwendungen auf der gemeinsamen Plattform geteilt. Ein erhöhter Energieverbrauch von einer Anwendung kann so die verfügbare Energie für eine andere Anwendung reduzieren. Dies gilt vor allem für batteriebetriebene Geräte [66]. Weiterhin kann eine hohe lokale Energiedichte und die damit verbundene erhöhte Temperatur benachbarte Komponenten auf demselben Chip beeinflussen oder sogar die Zuverlässigkeit und Lebensdauer des gesamten Chips reduzieren [79].

Sofern bei der geforderten Separierung die gemeinsame Nutzung von Ressourcen nicht komplett ausgeschlossen wird, ist es nicht nur wichtig zu erkennen, ob sich alle Anwendungen korrekt verhalten, sondern auch wie lange es dauert, eine eventuelle Abweichung von dem in der Analyse angenommenen Verhalten zu erkennen und gegebenenfalls weitere Abweichungen zu verhindern. Für eine Laufzeitüberwachung zur Separierung unterschiedlich kritischer Anwendungen, wie sie in in der vorliegenden Arbeit vorgestellte wird, ist es wichtig, eine Abweichung einer weniger kritischen Anwendung vom spezifizierten Verhalten zu erkennen und eine geeignete Reaktion abzuschließen, bevor sich die Abweichung fortpflanzt und zu einem Ausfall einer kritischen Anwendung führen kann. Eine mögliche Implementierung einer geeigneten Laufzeitüberwachung wäre, wie in den meisten aktuellen Systemen, durch eine zentrale Instanz, welche für Diagnosen, Selbsttest und Überwachung zuständig ist. Eine solche Lösung ist jedoch durch ihre mangelnde Skalierbarkeit nicht für Vielkernplattformen mit einer möglicherweise sehr großen Anzahl an zu überwachenden

Anwendungen geeignet. Mit der Größe des Systems und damit mit der Anzahl an zu überwachenden Komponenten würde die Reaktionszeit, um weitere Abweichungen vom spezifizierten Verhalten zu unterbinden, ansteigen. Durch eine längere Reaktionszeit könnten immer weniger Anwendungen auf einer gemeinsamen Plattform sicher separiert werden. Um die Reaktionszeit so kurz wie möglich zu halten und so eine Vielkernplattform optimal ausnutzen zu können, wird in dieser Arbeit eine dezentralisiert in Hardware implementierte Laufzeitüberwachung vorgestellt.

Die Laufzeitüberwachung ist Teil der größtenteils im Rahmen der vorliegenden Arbeit entwickelten *Integrated Dependable Architecture for Many Cores (IDAMC)*, einer parametrisierbaren und synthetisierbaren Vielkernplattform, die es erlaubt, unterschiedlich kritische Anwendungen zum einen transparent und flexibel auf einer gemeinsamen Plattform zu integrieren, und zum anderen, diese durch eine skalierbare Laufzeitüberwachung sicher zu separieren ohne die Vorteile eine Vielkernplattform übermäßig einzuschränken.

IDAMC besteht aus Kacheln (engl. *Tiles*), welche über ein NoC verbunden sind. Die Laufzeitüberwachung in den Netzwerkschnittstellen der einzelnen Kacheln stellt sicher, dass sich Fehler in einer der Kacheln nicht in dem restlichen System ausbreiten. Eine räumliche Separierung wird auf der IDAMC-Plattform daher über die Zuweisung von unterschiedlich kritischen Anwendungen zu unterschiedlichen Kacheln realisiert. Verteilte Ressourcen werden über eine transparente und flexible Adress- und Interrupt-Übersetzung, welche ebenfalls in den Netzwerkschnittstellen implementiert ist, zu einzelnen Anwendungen zugewiesen und vor Zugriffen durch andere Anwendungen geschützt. Die Zuordnung lässt sich hierbei auch zur Laufzeit ändern, ohne das betroffene Anwendungen hierfür angepasst werden müssen. Die flexible und transparente Zuordnung ermöglicht daher auch die Integration von bestehenden Anwendungen ohne Software-Änderungen. Benötigte Verbindungen des NoCs werden über virtuelle Kanäle räumlich voneinander separiert.

Die zeitliche Separierung auf der IDAMC-Plattform wird über eine geeignete Arbitrierung für jede gemeinsam verwendete Ressource sichergestellt und kann durch eine Analyse des zeitlichen Verhaltens verifiziert werden. Auf der Analyse des zeitlichen Verhaltens aufbauend wurde im Rahmen der vorliegenden Arbeit eine Analyse des Energieverbrauchs von einzelnen Anwendungen sowie der von diesen Anwendungen ausgehenden Energiedichte in einzelnen Chip-Regionen entwickelt. Der Einfluss von weniger oder nicht kritischen Aufgaben auf die Analyse sowohl des zeitlichen Verhaltens als auch des Energieverbrauchs von kritischen Aufgaben kann hierbei durch die Laufzeitüberwachung auf ein zuvor festgelegtes Maß begrenzt werden. Durch die kurze Reaktionszeit der vorgestellten Laufzeitüberwachung können unterschiedlich kritische Anwendungen sicher separiert werden und die Plattform dennoch effizient ausgenutzt werden.

Die vorliegende Arbeit ist wie folgt gegliedert. In Kapitel 2 liegt der Fokus auf Mehrprozessorsystemen und deren Eignung für die Integration von unterschiedlich kritischen Anwendungen. Es werden existierende Mehr- und Vielkernplattformen betrachtet sowie einzelne Mechanismen und Methoden zur transparenten, flexiblen und sicheren Ausführung von mehreren Anwendungen erläutert. Hierbei werden vor allem Ansätze und Besonderheiten zur Separierung, Laufzeitüberwachung, Kommunikation sowie Virtualisierung betrachtet. In Kapitel 3 wird zunächst der Einfluss von unterschiedlich kritischen Anwendungen auf die Analyse des zeitlichen Verhaltens eines Systems sowie die Reduzierung des Einflusses durch einen dynamischen Separierungsmechanismus beschrieben. Der Schwerpunkt liegt hierbei auf der gegenseitigen Beeinflussung durch eine kachelübergreifende Kommunikation zwischen Anwendungen und deren gemeinsamer Verwendung

von geteilten Ressourcen wie beispielsweise Speicherschnittstellen. Hierauf aufbauend wird die im Rahmen dieser Arbeit entwickelte Analyse des Energieverbrauchs von einzelnen Anwendungen beschrieben sowie die Bedeutung der Reaktionszeit einer Laufzeitüberwachung auf die Analyse erläutert. Kapitel 4 beschreibt dann die IDAMC-Vielkernplattform zur flexiblen, transparenten und sicheren Verwendung durch mehrere auch unterschiedlich kritische Anwendungen. Der Fokus liegt dabei auf den Netzwerkschnittstellen, welche die Kacheln und das NoC miteinander verbinden. Die darin enthalten neuartigen Hardware-Mechanismen zur Virtualisierung, zur kachelübergreifenden Kommunikation sowie zur dynamischen Separierung von unterschiedlich kritischen Anwendungen zur Laufzeit werden hierbei im Detail erläutert. Die Evaluierung der entwickelten Vielkernplattform auf einem *Field Programmable Gate Array (FPGA)* und ein Vergleich mit bestehenden Ansätzen, vor allem mit denen aus Kapitel 2, wird in Kapitel 5 beschrieben, bevor die Arbeit in Kapitel 6 zusammengefasst wird. Ein Ausblick auf Themen, die sich aus dieser Arbeit ergeben, wird ebenfalls in Kapitel 6 gegeben.

2 Mehrkernprozessoren

Das folgende Kapitel beschreibt zunächst Anforderungen an Mehrkernprozessoren für die Integration von unterschiedlich kritischen Anwendungen. Anschließend werden einige Mechanismen und Methoden erläutert, um diese Anforderungen zu erfüllen. Abschließend werden einige existierende Mehrkernplattformen beschrieben, welche es erlauben, mehrere Anwendungen zu integrieren, auch wenn einige davon eher nicht oder nur teilweise für den Einsatz in sicherheitsrelevanten Funktionen geeignet sind.

Auch wenn heutzutage in vielen Bereichen oft noch hoch optimierte und sehr komplexe Prozessoren mit einzelnen oder wenigen Rechenkernen mit teilweise hohen Frequenzen zum Einsatz kommen, ist bei Prozessoren seit einigen Jahren ein Trend in Richtung mehr, aber dafür weniger komplexe Rechenkerne mit niedrigeren Frequenzen erkennbar [87]. Die Erhöhung der Anzahl der Rechenkerne anstatt der Erhöhung der Rechenleistung von einzelnen Rechenkernen kann über die Regel von Pollack erklärt werden. Diese besagt, dass die Erhöhung der Rechenleistung eines Prozessors in etwa proportional zur Quadratwurzel der Erhöhung der damit einhergehenden Komplexität ist [22]. Eine Erhöhung der Komplexität bedeutet in diesem Fall eine Erhöhung der benötigten Chip-Fläche. Der Energieverbrauch eines Prozessors steigt dagegen in etwa linear mit der Komplexität an. Nach der Regel von Pollack benötigt man also für die Verdoppelung der Rechenleistung eines Einzelkernprozessors eine viermal so große Fläche und damit auch eine viermal so große Menge an Energie. Dagegen lässt sich eine Verdoppelung der Rechenleistung über die Verdoppelung der Anzahl der Rechenkerne bereits mit der doppelten Fläche und damit auch mit einem nur doppelt so hohen Energieverbrauch erreichen.

Dies trifft natürlich nur dann zu, wenn die Rechenleistung der zusätzlichen Rechenkerne auch komplett genutzt werden kann. Nach dem Amdahlschen Gesetz wird die Beschleunigung einer Anwendung und damit die effektive Nutzung von zusätzlichen Rechenkernen durch den Anteil an sequentiellem Code in der Anwendung begrenzt. Wenn die Beschleunigung durch eine Parallelisierung einer Anwendung niedrig ist, kann die Rechenleistung pro Watt sogar sinken, wenn eine Anwendung auf einer Vielkernplattform ausgeführt wird [93]. Dies wird darauf zurückgeführt, dass auch in den ungenutzten Rechenkernen durch Leckströme Energie verbraucht wird. Typischerweise laufen auf Vielkernplattformen jedoch mehr als eine Anwendung parallel, was der Rechenleistung pro Watt von Vielkernplattformen zugute kommt.

Ein weiterer Vorteil von Vielkernprozessoren ist die Möglichkeit, durch eine gleichmäßige Anordnung von Rechenkernen und Speichern eine gleichmäßige Temperaturverteilung auf dem Chip zu erhalten [49]. Ein über den Chip gleichmäßig verteilter Energieverbrauch verringert lokale Spitzentemperaturen, welche die Zuverlässigkeit des Chips reduzieren könnten. Weiterhin bieten viele Mehrkernprozessoren die Möglichkeit, die Rechenkerne mit individuellen Spannungen und Frequenzen zu betreiben sowie einzelne Rechenkerne abzuschalten, um Energie zu sparen.

Auf die vielfältigen Möglichkeiten der Kostenreduktion durch die Integration von mehreren Anwendungen auf einer einzelnen Rechenplattform wurde bereits in der Einleitung eingegangen.

Neben den genannten Vorteilen bei Kosten, Energieverbrauch, Volumen und Gewicht werden Entwickler durch die sinkende Verfügbarkeit von Einzelprozessoren in Zukunft möglicherweise gezwungen sein, auf Mehrkernprozessoren auszuweichen. Auch wenn hier die Möglichkeit besteht, alle zusätzlichen Rechenkerne zu deaktivieren, würde hiermit doch ein großes Potential ungenutzt bleiben.

2.1. Funktionale Sicherheit

Funktionale Sicherheit ist die Freiheit von inakzeptablen Risiken [51]. Risiko ist hierbei als das Produkt aus der Wahrscheinlichkeit für das Auftreten eines Schadens und der Schwere des Schadens definiert. Der Begriff Schaden umfasst dabei sowohl Verletzungen und Todesfälle bei Menschen als auch Sachschäden und Umweltschäden. Das Ziel ist es, das Restrisiko durch externe Maßnahmen, durch elektrische und elektronische oder auch durch mechanische Komponenten so weit zu reduzieren, dass es unter einer tolerierbaren Grenze liegt. Komplett vermeiden lassen sich Risiken in der Regeln nicht.

Für verschiedene Anwendungsbereiche existieren unterschiedliche Normen für die funktionale Sicherheit. Die IEC 61508 [51] ist eine Grundnorm für elektrische, elektronische und programmierbar elektronische Systeme, von welcher anwendungsspezifische Normen, wie beispielsweise die ISO 26262 [52] für den Automobilbereich, abgeleitet wurden. Die Normen beschreiben Anforderungen und Methoden zur Risikoreduzierung für den gesamten Lebenszyklus eines Produktes vom Entwurf bis zur Entsorgung. Das vorhandene Risiko, was von einem System ausgeht, muss zunächst ermittelt werden. Hiervon abhängig wird der Umfang der nötigen Risikoreduzierung in vier diskreten Stufen festgelegt. In der IEC 61508 werden diese Stufen *Sicherheitsintegritätslevel (SIL)*, mit Stufen von eins bis vier, und in der ISO 26262 *Automotive-Sicherheitsintegritätslevel (ASIL)*, mit Stufen von A bis D, genannt. Die Stufe vier beziehungsweise D ist hierbei die Stufe mit der höchsten geforderten Risikoreduzierung. Die in den Normen genannten Anforderungen und Methoden zielen zum einen auf das Vermeiden von systematischen Fehlern und zum anderen auf das Kontrollieren sowohl von systematischen als auch von zufälligen Fehlern. Die Anforderungen und damit der Aufwand für die Entwicklung eines System steigen mit höheren Stufen erheblich an, weshalb es zu vermeiden gilt, dass eine Funktion höhere Anforderungen erfüllen muss als unbedingt nötig.

Wenn jedoch ein System, welches mehrere Funktionen unterschiedlicher SILs oder auch sicherheitskritische und nicht sicherheitsbezogene Funktionen gemeinsam auf einer Plattform integriert, müssen sowohl die gesamte Hardware als auch die Software aller Funktionen die Anforderungen der Funktion mit dem höchsten SIL erfüllen. Hierdurch steigen vor allem die Kosten von weniger oder nicht kritischen Funktionen, aber auch die Kosten des Gesamtsystems stark an. Damit dies vermieden werden kann, muss eine ausreichende Separierung zwischen den unterschiedlich kritischen Funktionen sichergestellt werden [51]. Dies bedeutet, dass Ausfälle von weniger kritischen oder nicht sicherheitsbezogenen Funktionen nicht zu einem gefährlichen Ausfall einer kritischen Funktion führen dürfen. Es muss also nachgewiesen werden, dass die Wahrscheinlichkeit eines abhängigen Ausfalls zwischen unterschiedlich kritischen Aufgaben entsprechend den Anforderungen der höher kritischen Aufgaben ausreichend niedrig ist.

Abhängige Ausfälle sind Ausfälle, deren Wahrscheinlichkeiten sich nicht als einfaches Produkt der unbedingten Wahrscheinlichkeiten der einzelnen auslösenden Ereignisse darstellen lassen [51]. Zwei Ereignisse werden damit als abhängig bezeichnet, wenn $P(A \cap B) > P(A) \cdot P(B)$ gilt. Abhängige

Ausfälle lassen sich in Ausfälle mit gemeinsamer Ursache (engl. *Common Cause Failures*) und kaskadierende Fehler klassifizieren. Bei kaskadierenden Ausfällen zieht der Ausfall eines Teilsystems, den Ausfall weiterer Teilsysteme nach sich. Ausfälle mit gemeinsamer Ursache sind Ausfälle mehrerer unterschiedlicher Teilsysteme oder getrennter Kanäle, die gleichzeitig oder über eine Zeitraum auftreten, und ein einzelnes oder mehrere Ereignisse als gemeinsame Ursache haben. Hierzu gehören bei Mehrprozessorsystemen oft die Spannungsversorgung, Takt, Temperatur und elektromagnetische Einflüsse. Wenn einzelne Ursachen dazu führen können, dass mehrere sicherheitsrelevante Funktionen eines Systems ausfallen, kann dies, abhängig von dem hiervon ausgehenden Risiko, sogar dazu führen, dass alle betroffenen Funktionen, die Anforderungen eines höheren als für die einzelnen Funktionen festgelegten SILs erfüllen müssen [51]. Die Anfälligkeit eines Systems für Ausfälle mit gemeinsamer Ursache kann über die *Common Cause Failure Analysis* ermittelt und über den β-Faktor dargestellt werden. Ein niedriger β-Faktor deutet dabei auf eine geringere Anfälligkeit hin. Die Anfälligkeit von Mehrprozessorsystemen gegenüber Ausfällen mit gemeinsamer Ursache lässt sich beispielsweise über physisch getrennte Blöcke mit separaten Spannungsversorgungen oder einer diversitären Implementierung verringern. Die Anfälligkeit gegenüber Ausfällen mit gemeinsamer Ursache sowie die Robustheit der einzelnen Anwendungen ist jedoch nicht Kern der vorliegenden Arbeit.

Der Schwerpunkt dieser Arbeit liegt auf der Unabhängigkeit der Ausführung von unterschiedlich kritischen Anwendungen auf einer gemeinsamen Plattform. Unabhängigkeit der Ausführung zwischen Software-Elementen bedeutet hierbei, dass eine ungünstige gegenseitige Beeinflussung des Ausführungsverhaltens von Software-Elementen nicht zu einem gefährlichen Ausfall führt. Hierfür muss nachgewiesen werden, dass Anwendungen sowohl zeitlich als auch räumlich unabhängig voneinander sind oder dass jede Verletzung der Unabhängigkeit kontrollierbar ist [51]. Räumliche Trennung bedeutet, dass die Daten einer Anwendung nicht durch eine andere Anwendung verändert werden können, vor allem nicht durch eine nicht sicherheitsrelevante Anwendung. Durch eine zeitliche Trennung muss sichergestellt werden, dass eine Anwendung keine andere Anwendung an deren korrekter Ausführung hindert, indem sie zu viel der verfügbaren Rechenzeit eines Prozessors in Anspruch nimmt oder indem sie eine andere gemeinsam verwendete Ressource übermäßig lange blockiert. Hierbei müssen auch die oben genannten kaskadierenden Ausfälle betrachtet werden, um einen gefährlichen Ausfall einer kritischen Aufgabe durch eine negative Beeinflussung, ausgehend von einem fehlerhaften oder nicht spezifizierten Verhalten einer weniger kritischen oder nicht sicherheitsrelevanten Aufgabe, ausschließen zu können.

Bei Mehrkernprozessoren mit mehreren unterschiedlich kritischen Anwendungen muss also entweder eine gegenseitige räumliche und zeitliche Beeinflussung komplett ausgeschlossen oder kontrolliert werden können. Um eine ausreichende zeitliche Separierung durch eine Kontrolle zur Laufzeit sicherzustellen, muss ein abweichendes Verhalten einer Anwendung erkannt und unterbunden werden, bevor eine andere Anwendung soweit negativ beeinflusst wird, dass sie eine Frist verpasst und fehlschlägt. Hier lässt sich der Begriff der *Prozesssicherheitszeit* auf die einzelnen Anwendungen anwenden, welcher in der IEC 61508 definiert ist als Zeitraum zwischen einem Ausfall eines Systems, welcher zu einem gefährlichen Ereignis führen kann, und der Zeit, zu der eine angemessene Reaktion abschlossen sein muss, um das Auftreten des gefährlichen Ereignisses zu verhindern [51]. Anstatt des Ausfalls eines Systems würde man hier den Ausfall oder eine Abweichung

einer weniger kritischen Aufgabe betrachten und das zu verhindernde gefährliche Ereignis wäre die Fristüberschreitung einer kritischen Aufgabe auf demselben System.

Angewendet auf einen Vielkernprozessor mit mehreren auf unterschiedlichen Rechenkernen beziehungsweise Kacheln räumlich getrennten Anwendungen wären diese voneinander abhängig, wenn ein Ausfall einer Anwendung auf einer der Kacheln zu einem Fehler in einer anderen Kachel führen kann, welcher, wenn er nicht erkannt wird, den Ausfall der dort ausgeführten Anwendung nach sich zieht. Dies kann beispielsweise dadurch entstehen, dass eine fehlerhafte Aufgabe öfters als spezifiziert auf einen gemeinsam verwendeten Speicher zugreift, wodurch eine andere Aufgabe soweit verzögert wird, dass sie ihre Frist verpasst. In dem Beispiel muss also ein erhöhtes Aufkommen an Zugriffen auf einen gemeinsam verwendeten Speicher von einer weniger kritischen Aufgabe auf einer der Kacheln erkannt und weitere Zugriffe schnell genug unterbunden werden, bevor eine kritische Aufgabe auf einer anderen Kachel so weit verzögert wird, dass ihre Frist nicht mehr garantiert werden kann.

2.2. Separierung

Die meisten Mehr- und Vielkernprozessoren bieten eine gewisse Form der Separierung von unterschiedlichen Anwendungen an. Die Separierung verhindert das Fortpflanzen möglicher Fehler von einer in eine andere Anwendung. Weiterhin wird so eine unabhängige Entwicklung, Verifikation und Validierung der einzelnen Anwendungen ermöglicht [92]. Ohne ausreichende Trennung steigen die Kosten der Integration, besonders von komplexen Anwendungen, da in diesem Fall eine Verifikation und Validierung des Gesamtsystems erfolgen muss, wobei alle Anwendungen dann einheitlich den Ansprüchen der höchsten Kritikalitätsstufe entsprechen müssen. Eine ausreichende Separierung erlaubt die Entwicklung von Anwendungen durch unterschiedliche Lieferanten oder Teams, eine modulare Verifikation entsprechend den individuellen Kritikalitätsleveln sowie die spätere Änderung einzelner Partitionen.

Die eingesetzten Separierungsmechanismen können nach räumlicher und zeitlicher Trennung unterschieden werden. Einige legen die Trennung statisch für die gesamte Laufzeit fest und bei manchen Systemen kann die Separierung wiederum dynamisch angepasst werden. Eine sichere Partitionierung eines Systems und das effiziente Teilen von Ressourcen sind hierbei jedoch stets gegensätzliche Anforderungen [24]. Virtualisierung ist eine Technik, die es erlaubt, die verfügbaren Ressourcen sowohl zeitlich als auch räumlich sicher auf die einzelnen Anwendungen aufzuteilen. Verschiedene Formen der Virtualisierung werden in Abschnitt 2.4 genauer erläutert.

2.2.1. Räumliche Separierung

Weit verbreitete Mechanismen zur räumlichen Separierung von Zugriffen auf gemeinsam verwendete Speicher durch unterschiedliche Prozesse sind Speicherverwaltungseinheiten (engl. *Memory Management Units (MMUs)*) und Speicherschutzeinheiten (engl. *Memory Protection Units (MPUs)*). Ohne MMU oder MPU können Anwendungen auch auf den Speicherbereich anderer Anwendungen zugreifen und die darin enthaltene Daten verändern.

MPUs werden typischerweise durch das Betriebssystem konfiguriert und definieren Speicherbereiche, welche durch den aktuellen Prozess adressiert werden dürfen. Der aktuelle Prozess kann damit nicht mehr auf Speicherbereiche von anderen Prozessen zugreifen. Bei einem Kontextwechsel programmiert das Betriebssystem die MPU mit den Adressbereichen des nächsten Prozesses.

Eine MMU ist ein mächtigerer aber auch komplexerer Mechanismus als eine MPU. Eine MMU bietet einem Prozess Zugriff auf einen beliebigen virtuellen Adressraum. Die tatsächliche Zuordnung von virtuellen auf physische Adressen geschieht über Übersetzungstabellen, welche für jeden Prozess im Speicher abgelegt sind. Nach einem Kontextwechsel verweist die MMU auf die jeweils aktuelle Tabelle des nächsten Prozesses. Speicherbereiche, die nicht in der Übersetzungstabelle eines Prozesses enthalten sind, können nicht adressiert werden, wodurch auch mit einer MMU, wie mit einer MPU, eine räumliche Separierung von Speicherbereichen von unterschiedlichen Prozessen realisiert werden kann. Um die Übersetzung zu beschleunigen, verfügt eine MMU über *Translation Lookaside Buffers (TLBs)*, welche einen Teil der Übersetzungseinträge enthalten, zum Beispiel die am häufigsten verwendeten Einträge. Die Beschleunigung ist jedoch mit einer schlechteren zeitlichen Vorhersagbarkeit des Systems verbunden.

In Mehrprozessorsystemen mit individuellen Speicherschutzeinheiten für jeden Prozessor muss die Konfiguration der Einheiten der jeweils anderen Prozessoren fehlerfrei sein, was gerade bei Systemen mit unterschiedlich kritischen Anwendungen nicht als garantiert angenommen werden kann. Hattendorf et al. untersuchen in [41] verschiedene Varianten mit individuellen und geteilten Speicherschutzeinheiten. Die Autoren raten für Mehrprozessorsysteme mit unterschiedlich kritischen Anwendungen in jedem Fall zu einer gemeinsamen Speicherschutzeinheit und zwar explizit zu einer MPU, um eine zeitliche Analysierbarkeit des Systems sicherzustellen. Zusätzlich empfehlen die Autoren individuelle MPUs oder MMUs, je nach Anwendungsbereich der einzelnen Prozessoren. Einzelne Anwendungen ohne Betriebssystem kommen demnach ohne zusätzliche Speicherschutzeinheit aus, mehrere Anwendungen oberhalb eines Betriebssystems, welche eine zeitliche Analysierbarkeit erfordern, sollten auf einem Prozessor mit einer MPU implementiert werden und Anwendungen, welche die Mächtigkeit und Flexibilität einer MMU benötigen, sollten auf einem entsprechenden Prozessor mit MMU realisiert werden. Zeitliche Beeinflussungen durch eine übermäßige Anzahl an Zugriffen einer Anwendung auf einem der Prozessoren auf den eigenen Speicherbereich, jedoch unter Verwendung der gemeinsamen Speicherschnittstelle, werden in der vorgestellten Arbeit nicht behandelt.

Bei manchen Vielkernprozessoren konkurrieren Datenströme von unterschiedlichen Anwendungen bereits im NoC um Ressourcen und zwar um Puffer in den Switches sowie Verbindungen zwischen den Switches. Bei Flusskontrollmechanismen wie *Wormhole-Switching* oder *Virtual-Cut-Through-Switching*, bei denen sich die Teile eines Pakets schrittweise von den Puffern eines Switches zu den Puffern des nächsten Switches von der Quelle zur Senke bewegen, kann eine wechselseitige Blockierung von Puffern dazu führen, dass das gesamte System in eine Blockierung gerät. Dies kann vermieden werden, indem für unterschiedliche Datenströme unterschiedliche virtuelle Kanäle verwendet werden. Hierfür werden, wie bei Intels SCC [47], separate Puffer für unterschiedliche virtuelle Kanäle in den Switches implementiert. Die Verbindungen zwischen den Switches werden weiterhin geteilt und müssen über ein geeignetes Verfahren zeitlich aufgeteilt werden.

Bei Tileras TILE64-Prozessor [91] wurde ein anderer Ansatz gewählt. Hier wurden, anstatt virtueller Kanäle, fünf separate Netzwerk für unterschiedliche Arten von Datenverkehr implementiert, zwei Netzwerke für Speicherzugriffe, zwei Netzwerke für Anwendungen und ein Netzwerk für System- und I/O-Verwaltung. Die Autoren argumentieren, dass Verbindungen zwischen Switches mit den verfügbaren Ressourcen eines ASICs günstig zu realisieren sind, Puffer jedoch nicht. Da eine Implementierung mit separaten virtuellen Kanälen genauso viel Pufferplatz benötigt wie separate

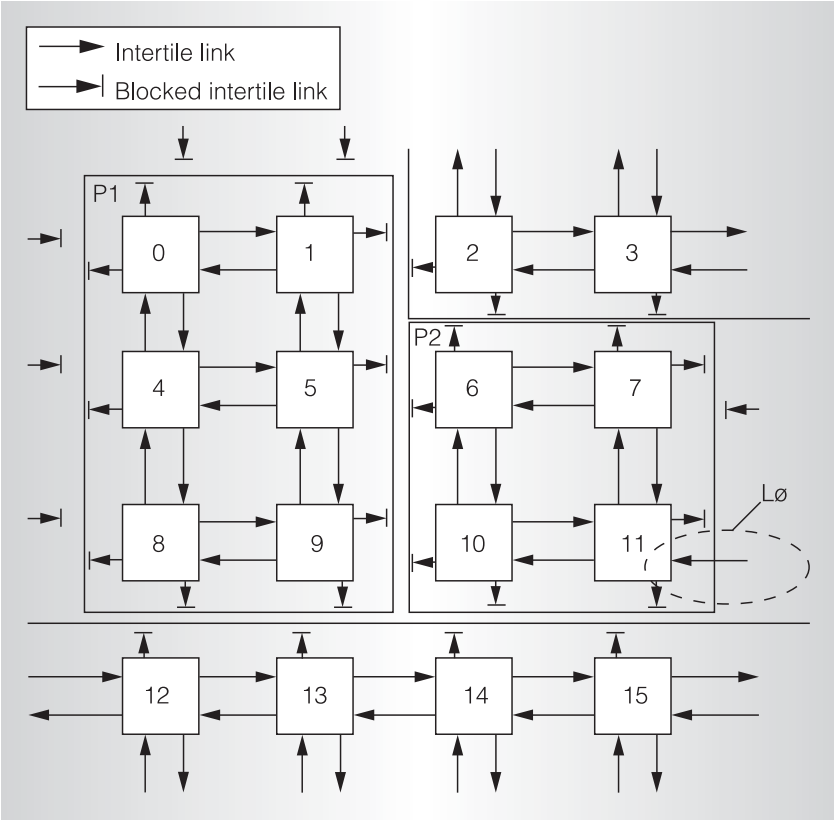

Abbildung 2.1.: Tileras Multicore Hardwall [91]

Netzwerke, welche darüber hinaus eine höhere Bandbreite durch separate Verbindungen ermöglichen, fiel die Entscheidung bei dem TILE64-Prozessor gegen den Einsatz von virtuellen Kanälen. Da die fünf vorhandenen Netzwerke alle einen bestimmten Zweck erfüllen, stehen bei einer Lösung mit dedizierten getrennten Netzwerken jedoch keine logisch voneinander trennbaren Netzwerke oder Kanäle für den Datenverkehr unterschiedlich kritischer Anwendungen zur Verfügung.

Die zwei für Anwendungen zugreifbare Netzwerke sowie das Netzwerk für System- und I/O-Verwaltung lassen sich durch einen zusätzlichen Separierungsmechanismus in mehrere Regionen aufteilen, wie in Abbildung 2.1 dargestellt. Die *Multicore Hardwall* von Tilera erlaubt es, alle Datenpakete über einzelne ausgehende Verbindungen eines Switches für die drei genannten Netzwerke zu blockieren. Blockierte Pakete werden in der lokalen Kachel über Interrupts signalisiert, um eventuell weitergehende Reaktionen auszulösen. Da jeweils ausgehende Verbindungen blockiert werden, lassen sich mit dem Mechanismus auch unidirektionale Verbindungen realisieren, wie in Abbildung 2.1 für den Switch mit der Nummer elf dargestellt, welche Pakete von kritischen in weniger kritische Region erlauben und trotzdem eine Fehlerfortpflanzung aus der weniger kritischen Region in die kritische Region verhindern können.

Aggarwal et al. präsentieren in [5] eine Plattform, die es erlaubt, eine größere Anzahl an gleichartigen Komponenten, wie Prozessoren P, Caches L/B und Speichersteuerungen auf unterschiedliche, voneinander isolierte, aber vollständige Teilsysteme, wie in Abbildung 2.2 dargestellt, aufzuteilen. Die Aufteilung geschieht über eine Trennung der gemeinsamen Ringverbindung aller Komponenten in mehrere Teilverbindungen, die jeweils ein fehlerisolierendes Teilsystem bilden. Jedes Teilsystem

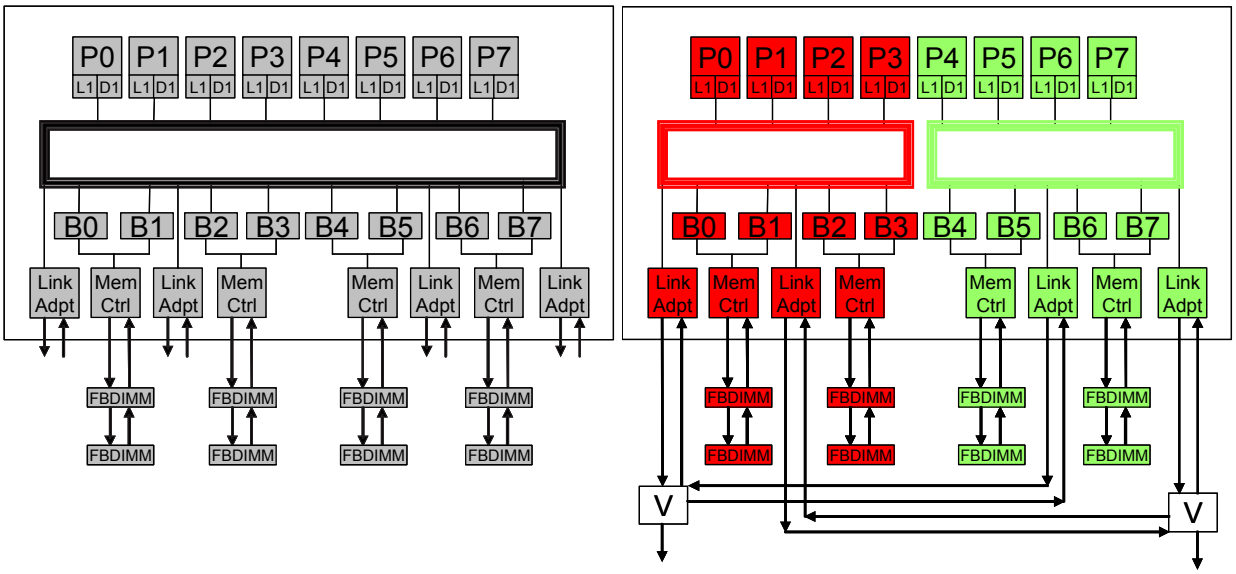

Abbildung 2.2.: Konfigurierbare Isolation [5]

kann so separat ausfallen, ohne die anderen Teilsystem zu beeinträchtigen. Dies wäre beispielsweise bei einer einfachen Verdoppelung des Prozessors bei einem busbasierten System nicht gegeben, da in dem Fall ein fehlerhafter Bus zu einem Ausfall des gesamten Systems führen würde. Die Konfigurierbarkeit der vorgestellten Lösung erlaubt es, komplett gemeinsam verwendete Systeme, wie in Abbildung 2.2 links dargestellt, oder auch unterschiedliche Formen der Redundanz zu generieren. So können mit derselben Plattform herkömmliche Systeme mit einer hoher Rechenleistung sowie hoch verfügbare oder hoch sichere System konfiguriert werden. Abbildung 2.2 rechts zeigt die parallele Ausführung derselben Anwendung (engl. *Dual Modular Redundancy (DMR)*), um deren Sicherheit zu erhöhen. Auch höhere Redundanzen wie *Triple Modular Redundancy (TMR)* oder Mischformen können konfiguriert werden. Der Vergleich von redundanten Teilsystemen geschieht über externe Vergleicher V an den Chip-Grenzen. Zusätzlich zu der Aufteilung in Regionen können einzelne Komponenten überwacht werden und im Fehlerfall deaktiviert werden, um betroffenen Anwendungen eventuell mit einem verringerten Funktionsumfang fortführen zu können. Die Konfiguration selbst und damit die Zuordnung von Komponenten zu Regionen lässt sich jedoch nicht zur Laufzeit ändern.

2.2.2. Zeitliche Separierung

Die zeitliche Separierung geschieht durch eine geeignete Ablaufsteuerung aller Aufgaben eines Systems, die eine geteilte Ressource verwenden. Auf Multiprozessorsystemen werden Aufgaben entweder global über alle vorhandenen Rechenkerne mit der Möglichkeit auf wechselnden Rechenkernen ausgeführt zu werden oder auf einzelne Rechenkerne begrenzt eingeplant. Eine lokale auf einzelne Prozessoren begrenzte Ablaufsteuerung ist sicherer aber weniger effizient, da Aufgaben hiermit besser isoliert sind aber nicht auf weniger ausgelastete Rechenkerne migriert werden können. Lokale Verfahren werden daher eher für hoch kritische Aufgaben verwendet. Globale Verfahren sind effizienter, führen jedoch zu mehr Konflikten in geteilten Caches und Netzwerken [63].

Die zeitlich Ablaufsteuerung der einzelnen Aufgaben eines Systems kann entweder zeitgesteuert oder ereignisgesteuert erfolgen. Auch Mischformen sind möglich. Bei zeitgesteuerten (engl. *time-triggered*) Architekturen werden Ressourcen nach einem festen periodisch wiederkehrenden Zeitplan aufgeteilt. Die Zuweisung von Ressourcen geschieht nur zu fest vorgegebenen Zeitpunkten. Wenn Anwendungen eine Ressource nicht den gesamten ihnen zugewiesenen Zeitschlitz benötigen, können Ressourcen bei dem Verfahren auch ungenutzt bleiben. Das fehlende reaktive Verhalten einer zeitgesteuerten Architektur kann zu einer weiteren Überdimensionierung des Systems führen, da beispielsweise Sensordaten viel häufiger ausgelesen werden müssen, wenn eine dynamische Reaktion vermieden werden soll [23]. Eine zeitgesteuerte Ablaufplanung nutzt die Ressourcen eines Systems daher in der Regel schlechter aus als eine ereignisgesteuerte Ablaufplanung, benötigt dafür aber keine aufwendige Analyse.

Bei der ereignisgesteuerten Ablaufplanung geschieht die Zuweisung nicht zu festgelegen Zeitpunkten sondern bei bestimmten Ereignissen, wie zum Beispiel Interrupts. Ressourcen werden dann an Aufgaben entsprechend ihrer Priorität zugewiesen. Die Prioritäten von Aufgaben können statisch festgelegt sein oder sich zur Laufzeit dynamisch ändern. Die ereignisgesteuerte Ablaufsteuerung lässt Ressourcen nicht ungenutzt, erfordert jedoch einen teils erheblichen Analyseaufwand. Mit einer hierarchischen Ablaufsteuerung lassen sich auch Mischformen realisieren, indem beispielsweise feste Zeitschlitze an Anwendungen zugewiesen werden in denen dann die einzelnen Aufgaben der Anwendung ereignisbasiert gesteuert werden können [24].

Das Zeitverhalten von Aufgaben auf einem System mit unterschiedlich kritischen Anwendungen ist durch ihre Aktivierungsperiode, Frist, Ausführungszeit und Kritikalität definiert [24]. Da bei der zeitlichen Planung von weniger kritischen Aufgaben die zu betrachtenden Ausführungszeiten von höher kritischen Aufgaben unnötig pessimistisch sind, kann man unterschiedliche Ausführungszeiten für unterschiedliche Stufen der Kritikalität verwenden [63]. Die angenommenen Ausführungszeiten der einzelnen relevanten Aufgaben sind dann umso niedrigen, desto niedriger die Kritikalität der gerade betrachteten Aufgabe ist.

Damit sich unterschiedlich kritische Aufgaben nicht gegenseitig beeinflussen, dürfen Aufgaben nicht öfter als angenommen aktiviert werden und nach einer Aktivierung nicht länger als definiert ausgeführt werden. Umso kritischer die Aufgaben eines Systems sind, desto konservativer muss deren Verhalten definiert werden. Dies bedeutet, dass bei Systemen mit kritischen Aufgaben eher von einer öfteren Aktivierung und einer längeren Ausführungszeit ausgegangen wird. Hierdurch verlängern sich die angenommenen Antwortzeiten für den schlechtesten Fall (engl. *Worst-Case Response Time (WCRT)*) vor allem für kritische Aufgaben. Aus Zertifizierungs- und Validierungssicht kann ein System damit komplett ausgelastet sein. Da hoch kritische Aufgaben auf dem realen System jedoch aufgrund des angewendeten Pessimismus meist nur wenig von ihrer zugedachten Zeit verwenden, kann hiermit viel Zeit ungenutzt bleiben (engl. *Slack Time*) [63].

Es gibt verschiedenen Ansätze, den hierbei entstehenden Slack für weniger kritische Aufgaben zu verwenden, indem diese beispielsweise ausschließlich in dem Slack von kritischen Aufgaben ausgeführt werden [24]. Weiterhin ist es möglich, weniger kritische Aufgaben zu bevorzugen, solange die Fristen der kritischen Aufgaben garantiert werden können, um so den Gesamtdurchsatz zu erhöhen. Dies lässt sich beispielsweise so realisieren, dass weniger kritische Aufgaben nach Beendigung einer kritischen Aufgabe in der sowieso eingeplanten verbliebenen Zeit ausgeführt werden, auch wenn andere kritische Aufgaben ebenfalls aktiviert werden könnten [63]. Da diese Zeit bereits eingeplant

war, kann ausgeschlossen werden, dass hierdurch eine andere hoch kritische Aufgaben ihre Frist verpasst.

Die Autoren von [27, 55, 23] untersuchten Mehrkernsysteme auf gegenseitige zeitliche Beeinflussungen von unterschiedlichen Anwendungen und erarbeiteten Vorschläge, diese Beeinflussung zu verhindert oder wenigstens abzuschwächen. Da viele Ressourcen auf einem Mehrkernsystem nicht mehr exklusiv von einzelnen Anwendungen oder einzelnen Betriebssystemen verwendet werden, entstehen Konflikte um oft komplex arbitrierte Ressourcen, was zu einem nichtdeterministischen oder schwierig zu analysierenden Zeitverhalten führen kann. Zugriffe auf gemeinsam verwendete Ressourcen müssen daher für ein deterministisches Verhalten entweder verhindert oder kontrolliert werden. Alternativ muss jedes mögliche Verhalten für eine Abschätzung der WCRT aller kritischen Aufgaben einbezogen werden, was zu sehr pessimistischen Antwortzeiten führen kann [55].

Cilku et al. [27] argumentieren, dass ein Hypervisor zwar eine logische Trennung zwischen unterschiedlichen Partitionen herstellen kann, sich deren Zeitverhalten durch die gemeinsame Verwendung von physischen Ressourcen aber immer noch gegenseitig beeinflussen kann, selbst wenn diese nicht miteinander kommunizieren. Wo bei komplexen Einzelprozessoren durch *Simultaneous Multithreading (SMT)* Konkurrenzsituationen um Pipeline-Ressourcen vorhanden sind, entstehen bei Mehrkernprozessoren mit *Symmetric Multiprocessing (SMP)* Konflikte um geteilte Caches, Busse oder Switches eines NoCs, externe Speicher oder I/O-Schnittstellen. Hierdurch wird es immer schwieriger sicherzustellen, dass reale Anwendungen ihrem spezifizierten Zeitverhalten entsprechen [23]. Bei SMP-Systemen kann, durch ein erhöhtes Interrupt-Aufkommen für die Kommunikation und Synchronisierung zwischen den einzelnen Rechenkernen, die gegenseitige zeitliche Beeinflussung von Anwendungen sogar noch weiter verstärkt werden [55].

Werden mehrere virtuelle Rechenkerne auf einen einzelnen physischen Rechenkern abgebildet, teilen sich die virtuellen Rechenkerne auch einen einzelnen physischen Cache [27]. Das Zeitverhalten einer Partition hängt damit davon ab, wie die jeweils anderen Partitionen den Cache verwendet. Ein Cache-Block kann nach einem Kontextwechsel durch die andere Partition ersetzt worden sein oder auch nicht. In jedem Fall entsteht hierdurch ein Nichtdeterminismus. Dasselbe gilt auch für höhere Cache-Ebenen, welche von mehreren physischen Rechenkernen gemeinsam verwendet werden. Die Autoren schlagen hierfür das Aufteilen des Caches mit Hardware- oder Software-Mechanismen vor, um für jede Partition einen exklusiv verwendeten Cache zu generieren. Für gemeinsam von mehreren virtuellen Rechenkernen verwendete TLBs ist die Argumentation für Caches ebenso gültig, weshalb auch hier eine Aufteilung das zeitliche Verhalten der ausgeführten Anwendungen besser vorhersagbar machen würde.

Selbst bei privat genutzten Caches können Schreibzugriffe auf einen von mehreren Rechenkernen gemeinsam verwendeten kohärenten Speicherbereich eine Aktualisierung der privaten Caches der anderen Rechenkerne erforderlich machen, was bei den dort ausgeführten Anwendungen wieder zu einem Nichtdeterminismus führen kann. Die für die Aktualisierung der privaten Caches der anderen Rechenkerne benötigten Zugriffe auf den gemeinsamen Bus können zu einer weiteren gegenseitigen zeitlichen Beeinflussung führen [55]. Alternativ zur Verwendung von Caches können kleine private durch Software kontrollierte Speicher für einen expliziten Nachrichtenaustausch zwischen Anwendungen auf unterschiedlichen Rechenkernen eingesetzt werden [55, 23].

Buszugriffe von unterschiedlichen Partitionen sind eine weitere Quelle von gegenseitiger zeitlicher Beeinflussung. Wenn Partitionen auf unterschiedlichen physischen Rechenkernen ausgeführt

werden, kann es zu Konflikten beim Zugriff auf den gemeinsam verwendeten physischen Bus kommen [27]. Diese werden durch Cache-Fehltreffer und Kohärenzprotokolle, DMA-Zugriffe oder andere Bus-Master noch weiter verstärkt und damit schwieriger zu analysieren [55]. Der verwendete Bus-Arbiter kann die Zugriffe trennen, indem er allen Busteilnehmern feste Zeitscheiben für deren Zugriffe zuteilt (engl. *Time Division Multiple Access (TDMA)*). Jede Partition darf dann nur noch in der ihr zugewiesenen Zeitscheibe auf den Bus zugreifen [55, 27]. Hierdurch ist es jedoch möglich, dass Buskapazitäten ungenutzt bleiben, weshalb Cilku et al. vorschlagen, einen Teil der statischen Zeitschlitze zusammenzufassen und für eine dynamische Arbitrierung von weniger kritischen Aufgaben zu verwenden.

Die Kommunikation über ein NoC kann ebenfalls zu einem zeitlichen Nichtdeterminismus führen, da das zeitliche Verhalten hierbei von der aktuellen Netzauslastung und von Routing-Entscheidungen in den Switches abhängt. Eine prioritätenbasierte Steuerung des NoCs kommt daher nicht ohne eine aufwendige Analyse aus. Unter Verwendung des TDMA-Verfahrens mit einer globalen Ablaufplanung und einer eventuellen Verzögerung von Paketen an der Quelle kann ein konfliktfreies Routing ermöglicht werden. Eine statische Ablaufplanung kann jedoch nicht auf Veränderungen des Systems zur Laufzeit reagieren und eine dynamische Änderung der globalen Ablaufplanung ist nicht skalierbar für große Systeme [23].

Die Zugriffszeiten auf externe DRAMs können ebenfalls zu zeitlichen Koppelungen zwischen Anwendungen führen, da diese von vorangegangenen Zugriffen und Auffrischungszyklen des Speichers abhängen [27, 55]. Wird beispielsweise erneut von derselben bereits geöffneten Reihe gelesen, geschieht dies deutlich schneller als das Lesen von eine Reihe, die zunächst geladen werden muss. Das Schließen jeder Reihe nach einem Zugriff und das damit nötige Neuladen einer Reihe bei jedem weiteren Zugriff würde ein deterministisches Verhalten erzeugen, jedoch auf Kosten der mittleren Zugriffszeit. Weiterhin könnten Auffrischungszyklen bei der zeitlichen Ablaufplanung mit einbezogen werden, um so gleichzeitige konkurrierende Speicherzugriffe von Anwendungen zu unterbinden. Damit das Zeitverhalten einer Anwendung nicht von externen Speicherzugriffen anderer Anwendung abhängt, schlagen die Autoren eine Zuweisung von festen Speicherbänken an jede Partition vor [27, 55].

Um das zeitliche Verhalten von Anwendungen, welche in einer Hochsprache geschrieben wurden, analysieren zu können, werden detaillierte Hardware-Modelle benötigt. Die Ausführung von Anwendungen auf unterschiedlichen *Instruction Set Architectures (ISAs)* und selbst mehrere Läufe auf derselben ISA können ein unterschiedliches Zeitverhalten aufweisen. Deren Gründe bleiben einem Anwendungsentwickler verborgen, da das Zeitverhalten der Mikroarchitektur durch die ISA vor höheren Ebenen verborgen wird und weder der Binärcode noch C-Programme eine Zeitsemantik beinhalten [23]. Eine grobe Isolierung des Zeitverhaltens von unterschiedlichen Anwendungen wird durch teils sehr aufwendige Hardware-Mechanismen ermöglicht. Dies ist durch die fehlende Zeitsemantik in der ISA nötig, erschwert jedoch eine feine Steuerung des Zeitverhaltens durch höhere Software-Ebenen.

Bui et al. [23] schlagen daher die Einführung einer Zeitsemantik für die ISA und Instruktionen für die Kontrolle des zeitlichen Verhaltens vor. Mit diesen Instruktionen sollen dann Anwendungen in einer Hochsprache mit der Definition des zeitlichen Verhaltens geschrieben werden können, ohne dabei tiefere Effekte in der Hardware beachten oder nachträglich analysieren zu müssen. Auch Modelle des Zeitverhaltens einer Architektur könnten damit auf höheren Ebenen generiert

werden. Mit den neuen Instruktionen sollen die Mindest- und Maximalausführungszeiten eines Programmblocks definiert werden können sowie Verzweigungen am Ende oder innerhalb eines Blockes bei Überschreitung der Maximalausführungszeit ermöglicht werden. Der Compiler generiert dann, durch eine statische Analyse aller Pfade, Anforderungen für die Hardware, damit alle Fristen eingehalten werden können. Der Linker kann dann diese Anforderungen mit den Fähigkeiten der Hardware abgleichen und kritische Anwendungen gegebenenfalls ablehnen.

2.3. Laufzeitüberwachung

Um den Zustand eines Mehr- oder Vielkernprozessorsystems sowie das Verhalten der ausgeführten Anwendungen zur Laufzeit überwachen zu können, sind die meisten Systeme mit erweiterten Überwachungsmechanismen ausgestattet [3]. Diese werden zum Debugging [28, 88, 36], für Optimierungen [36, 34], zur Evaluierung [31], zur Fehlererkennung [34] sowie zur Fehlerisolierung [43] eingesetzt.

In [88] wird eine Laufzeitüberwachung eingesetzt, um die Beobachtbarkeit und Steuerbarkeit des Systems für das Debugging der Rechenleistung zu erhöhen. Die eingefügten Überwachungsmodulen werden über den Testport des Systems ausgelesen und extern ausgewertet. Die überwachten Transaktionen werden jedoch bereits auf dem Chip vorgefiltert, um die Menge der auszulesenden Daten zu reduzieren.

Ein weiteres System, in dem die Überwachung nicht direkt für dynamische Änderungen des Systems zur Laufzeit verwendet wird, sondern zur Evaluation von Designalternativen unter Berücksichtigung bestimmter Systemeigenschaften, wie Energieverbrauch oder Rechenleistung, wird in [31] vorgestellt. Dedizierte Überwachungsmodule zählen Ereignisse und erfassen Statistiken, beispielsweise über Cache-Fehltreffer, um so eine optimale Konfiguration von Prozessoren, Speicherhierarchien und Verbindungen zu ermitteln.

Während der Entwicklungsphase erlaubt der Überwachungsmechanismus von Ciordas et al. [28] die Platzierung von Überwachungsinstanzen an beliebigen Routern oder Netzwerkschnittstellen des eingesetzten NoCs. Diese Instanzen können zur Laufzeit programmiert werden, um die zu überwachenden Eigenschaften auszuwählen und so die Netzwerkbelastung durch Überwachungspakete reduzieren zu können. Die Überwachung geschieht ohne Beeinflussung des zu überwachenden Datenverkehrs und dient als Grundlage für das Debugging. Die gesammelten Daten können hierfür zentral oder dezentral verarbeitet werden. Durch die hierfür benötigte Zeit ist der vorgestellte Mechanismus für die Reaktion auf ein abweichendes Verhalten von unterschiedlich kritischen Anwendungen zur Laufzeit jedoch nicht geeignet.

In [34] werden ebenfalls Überwachungsinstanzen zu den Routern eines NoCs hinzugefügt. Hier werden die Daten jedoch nicht für das Debugging gesammelt, sondern um Fehler im Netzwerk zu erkennen und darauf gegebenenfalls zur Laufzeit zu reagieren. Bei erkannten Fehlern, wie beispielsweise einer zu langen Laufzeit eines Paketes im NoC, wird der Sender benachrichtigt. Daraufhin kann das Paket erneut gesendet werden oder die Anwendung auf andere Ressourcen migriert werden. Durch die möglicherweise lange Reaktionszeit durch das erforderliche Senden von Nachrichten durch das NoC bei erkannten Abweichungen, ist die Lösung jedoch nur eingeschränkt für kritische Anwendungen geeignet. Die Separierung von unterschiedlich kritischen Aufgaben wird in [34] nicht weiter behandelt.

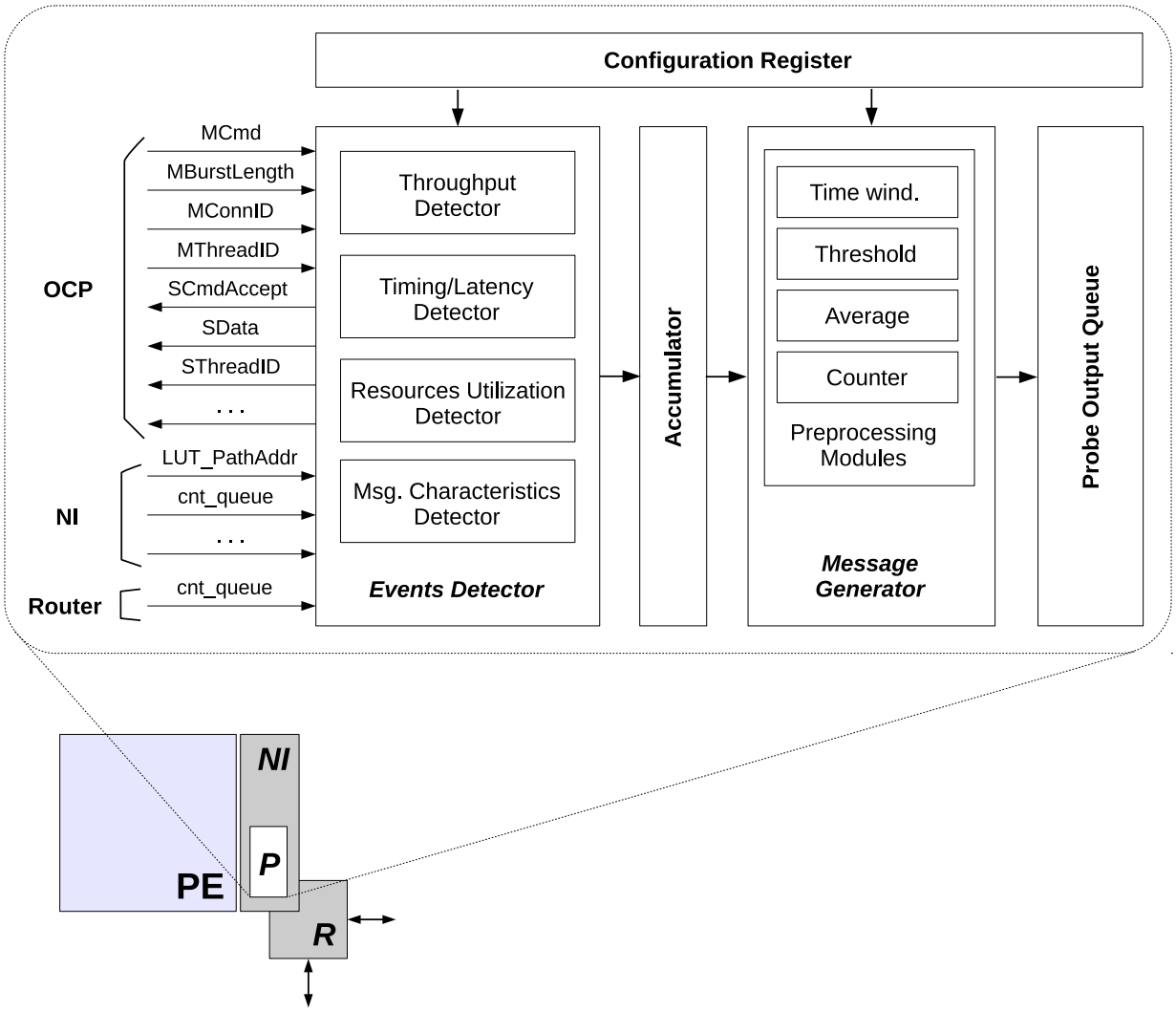

Abbildung 2.3.: Überwachungsmechanismus in den Netzwerkschnittstellen [36]

Die Autoren von [36] stellen eine flexible und konfigurierbare Überwachungslösung für den von einer Kachel ausgehenden Datenverkehr vor. Die Vielseitigkeit der Überwachungsinstanzen erfordert jedoch einen hohen Verbrauch an Hardware-Ressourcen, was die Lösung für sehr große Systeme mit vielen Verbindungen nur sehr eingeschränkt einsetzbar macht. Der Mechanismus ist, wie in Abbildung 2.3 dargestellt, in den Netzwerkschnittstellen der einzelnen Rechenelemente PE implementiert. Die gesammelten Daten können für Optimierungen, Debugging oder die Ermittlung von Eigenschaften der ausgeführten Anwendungen eingesetzt werden. Die Überwachungsinstanzen erlauben unter anderem die Überwachung der Verwendung von verteilten Komponenten sowie die Menge an ausgehendem Datenverkehr von den einzelnen Kacheln. Die Konfiguration der Laufzeitüberwachung in den Netzwerkschnittstellen wird von einer zentralen Einheit durchgeführt, welche auch für die weitere Verarbeitung der gesammelten Daten zuständig ist. Um die Anzahl der Nachrichten an die zentrale Überwachungseinheit zu reduzieren, können die registrierten Ereignisse in den Netzwerkschnittstellen vorverarbeitet werden. Durch fehlende automatische Reaktionen und die benötigte Interaktion mit der zentralen Überwachungsinstanz würde die Reaktionszeit auf

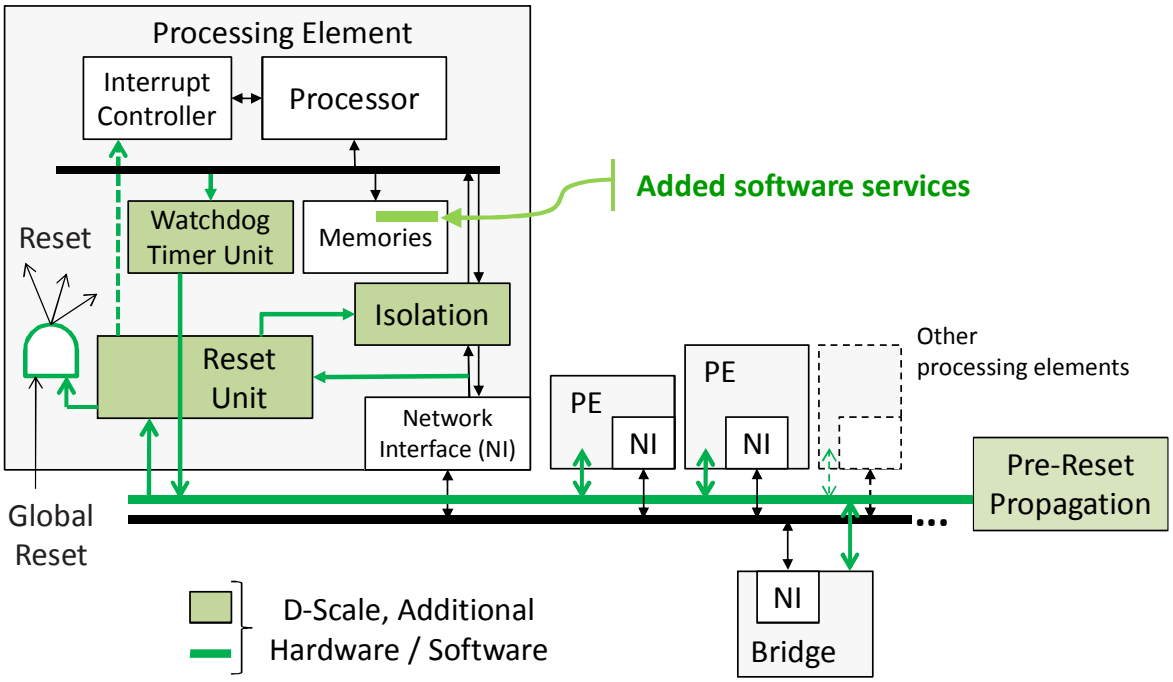

Abbildung 2.4.: D-Scale-Architektur [42]

ein erkanntes Fehlverhalten einer Anwendung bei der vorgestellten Lösung jedoch mit der Größe des Systems ansteigen. Daher ist der präsentierte Mechanismus in der dargelegten Form nicht für eine effiziente Separierung von unterschiedlich kritischen Aufgaben auf einer Vielkernplattform einsetzbar.

Ein Mechanismus, welcher über eine gegenseitige Überwachung der einzelnen Rechenelemente einer Mehrkernplattform die Isolierung von fehlerhaften Elementen ermöglichen soll, ist in [43] beschrieben. Die in Abbildung 2.4 dargestellte D-Scale-Architektur besteht aus mehreren Rechenelementen PE, welche über ein beliebiges Netzwerk verbunden sind. Einige Rechenelemente können als Ersatzelemente vorgehalten werden, um die Verfügbarkeit des Systems zu erhöhen. Jedes Rechenelement enthält, wie dargestellt, eine *Watchdog*-Schaltung, welche periodisch zurückgesetzt werden muss. Die Rücksetzung geschieht in einer Kette, jeweils über das benachbarte Rechenelement. Bleibt die Rücksetzung aus, wird von einem Fehler in dem für die Rücksetzung verantwortlichen benachbarten Element ausgegangen, woraufhin sämtliche Elemente isoliert werden und das gesamte System neu gestartet wird. Für die Periode der Rücksetzung wurden der benötigte Rechenmehraufwand für die Rücksetzung und die resultierende Fehlererkennungszeit gegenübergestellt. Der Neustart des gesamten Systems wurde aufgrund der geringen Diagnosemöglichkeiten einer *Watchdog*-Schaltung sowie der schnellen Ausbreitung von Fehlern im Netzwerk gewählt. Vor dem Neustart des Systems können alle Rechenelemente die nötigsten Daten für die Zeit nach dem Neustart in einem nicht von der Rücksetzung betroffenen Speicher ablegen. Mit den gespeicherten Daten kann nach dem Reset dann beispielsweise eine Migration von einer Anwendungen auf ein anderes Rechenelement eingeleitet werden. Auch wenn alle Anwendungen des Systems nach einem Neustart eventuell wieder fehlerfrei ausgeführt werden können, sind die Anwendungen der Plattform voneinander abhängig, da ein Fehler in einem Rechenelement zu einem Neustart des gesamten Systems führt.

2.4. Virtualisierung

Virtualisierung wird traditionell dafür eingesetzt, um Systeme besser auszulasten, indem sie mehrere virtuelle Maschinen isoliert voneinander auf einer einzelnen physischen Plattform ermöglicht. Weiterhin bietet die Virtualisierung eine Entkoppelung von der tatsächlich vorhandenen Hardware und erlaubt so die Verwendung von Anwendungen oder Betriebssystemen, die sonst nicht direkt auf der physischen Plattform ausgeführt werden könnten. Dies ist besonders interessant für bestehende Anwendungen, auch *Legacy*-Anwendungen genannt, welche auf einem neueren System weiterverwendet werden sollen. Ursprünglich wurde die Technologie entwickelt, um die verfügbare Rechenleistung von Serverplattformen effizient auszunutzen. Aufgrund immer weiter steigender Rechenleistung von Prozessoren ist Virtualisierung nun ebenfalls im Anwenderbereich Standard und wird zunehmend auch in eingebetteten Systemen eingesetzt.

Virtualisierung ist in der Regel eine Software-Schicht, welche *Hypervisor* oder *Virtual Machine Monitor (VMM)* genannt wird und zwischen der tatsächlichen Hardware und den Gastsystemen liegt. Der Hypervisor abstrahiert von der realen Hardware und übersetzt Zugriffe auf virtuelle Ressourcen in Zugriffe auf physische Ressourcen oder emuliert diese. Physische Ressourcen können exklusiv an Gastsysteme gebunden werden oder von mehrere Gastsystemen geteilt werden, indem sie jeweils nur zeitweise von den einzelnen Gastsystemen verwendet werden können. Die Gastsysteme haben jedoch stets die Sicht auf eine exklusiv benutzte Hardware mit allen benötigten Komponenten und Zugriffsrechten, wodurch deren Entwicklung vereinfacht wird. Die Zuordnung von Ressourcen lässt sich grundsätzlich auch zur Laufzeit flexibel und für die Gastsysteme transparent ändern, wodurch die Virtualisierung auch zur Optimierung oder Erhöhung der Fehlertoleranz eines Systems beitragen kann.

Zwei weit verbreitete Varianten der Virtualisierung sind die vollständige Virtualisierung (engl. *Full Virtualization*) und die Paravirtualisierung. Normalerweise existieren auf Systemen zwei Berechtigungsstufen, der Kernelbereich für das Betriebssystem mit allen Rechten und der Nutzerbereich mit eingeschränkten Rechten für die Anwendungen. Wenn nun durch die Virtualisierung eine weitere Software-Schicht eingeführt wird, benötigt die vollständige Virtualisierung spezielle Hardware-Unterstützung, um ausschließlich dem Hypervisor kompletten Zugriff auf alle Ressourcen zu gewähren und weiterhin die Gastbetriebssysteme und darauf laufenden Anwendungen auf unterschiedlichen Berechtigungsstufen auszuführen. Ist diese spezielle Hardware-Unterstützung vorhanden, ermöglicht die vollständige Virtualisierung die Ausführung der Gastsysteme ohne Modifikationen.

Da in den meisten eingebetteten Systemen die benötigte Hardware-Unterstützung jedoch nicht vorhanden ist, bietet sich die Paravirtualisierung als Alternative an. Da hier dann nur zwei Berechtigungsstufen vorhanden sind, läuft nur der Hypervisor im Kernelbereich mit allen Rechten und die Gastbetriebssysteme und deren Anwendungen zusammen im Nutzerbereich mit eingeschränkten Rechten [59]. Sensitive Instruktionen der Gastbetriebssysteme, die im Nutzerbereich dann nicht mehr erlaubt sind, müssen manuell in der Software durch *Hypercalls* ersetzt werden, welche von der Virtualisierungsschicht bereitgestellt werden. Dieser Aufwand der Software-Änderung muss bei jeder neuen Version des Gastbetriebssystems wiederholt werden. Falls der Quelltext des Gastsystems nicht verfügbar ist, muss der Objektcode zur Laufzeit geändert werden, um sensitive Befehle durch Sprünge in die Virtualisierungsschicht zu ersetzten [81].

So wie ein Betriebssystem die verschiedenen auszuführenden Aufgaben voneinander separiert, separiert ein Hypervisor die unterschiedlichen Gastsysteme voneinander und kann deren Verhalten überwachen. Kommunikation zwischen den Gastsystemen ist nur über den Hypervisor möglich. Durch Schwachstellen im Hypervisor ist es jedoch möglich, dass sich Fehler oder bewusste Manipulationsversuche aus einem Gastsystem auf den Hypervisor und andere Gastsysteme auswirken. Daher ist es erstrebenswert, den Umfang eines Hypervisors möglichst gering zu halten, um so dessen Sicherheit zu erhöhen.

Mit NOVA [81] wurde versucht, den Umfang der kritischen Basisfunktionalität des Hypervisors (engl. *Trusted Computing Base*) so weit wie möglich zu reduzieren. NOVA besteht aus kleinen und einfachen Teilen, die unabhängig voneinander verifiziert werden können. Nur was kritisch in Bezug auf Sicherheit oder Geschwindigkeit ist, wird im Kernelbereich ausgeführt. Die Virtualisierung selbst wird bei NOVA im Nutzerbereich realisiert. Die Autoren trennen daher, anders als üblich, die Begriffe Hypervisor und Virtual Machine Monitor. Der Hypervisor, oder auch *Microhypervisor* [81], läuft privilegiert im Kernelbereich und kontrolliert MMUs, eventuell vorhandenen IOMMUs, Interrupt-Steuerungen und Zeitgeber. Alles andere, wie Gerätetreiber, Dateisystem und ein VMM pro Gastsystem, um zwischen virtuellen Maschinen und tatsächlicher Hardware zu übersetzen, laufen im Nutzerbereich. NOVA ist auf Hardware-Unterstützung angewiesen, ermöglicht so aber unmodifizierte Gastsysteme.

Neben Hypervisoren werden auch *Microkernel* für eingebettete Systeme eingesetzt. Sowohl Hypervisoren als auch Microkernel dienen als Grundlage für größere Systeme, jedoch mit unterschiedlichen Zielen [44]. Microkernel implementieren nur die nötigsten Mechanismen im Kernelbereich, wie die Erzeugung von Aufgaben, die zeitliche Aufteilung von Ressourcen sowie die Kommunikation zwischen Prozessen, um so den Umfang der kritischen Basisfunktionalität der Software möglichst klein zu halten, damit sich deren Sicherheit über formale Beweise verifizieren lässt. Alle anderen nicht zwingend notwendigen Dienste, wie ein Dateisystem oder Gerätetreiber, werden auf Nutzerebene implementiert. Mechanismus und Methode werden bei Microkerneln getrennt. Zusammen können sie zur Konstruktion beliebiger Systeme verwendet werden.

Auch wenn diese Eigenschaften in großen Teilen ebenfalls für den Hypervisor NOVA [81] zutreffen, ist die eigentliche Motivation hinter Hypervisoren, die Weiterverwendung von bestehenden Anwendungen und Betriebssystemen sowie die nebenläufige Ausführung von unterschiedlichen Systemen auf mehreren voneinander isolierten virtuellen Maschinen [44]. Diese sollten möglichst echt wirken, wobei Microkernel möglichst klein sein sollten. Die Trennung zwischen Hypervisoren und Microkerneln verschwindet jedoch immer mehr [44]. Hypervisoren werden, wie an dem *Microhypervisor* NOVA [81] zu sehen, mehr wie Microkernel, indem sie immer schlanker und dadurch sicherer werden und indem sie Gerätetreiber in den Nutzerbereich verlagern, um so Standardtreiber verwenden zu können. Microkernel werden dafür den Hypervisoren immer ähnlicher, indem sie, wie der *Microvisor* OKL4 [44], immer mehr Virtualisierung für bestehende Anwendungen unterstützen. OKL4 kommt ohne Hardware-Unterstützung aus, erfordert jedoch, wie bei Paravirtualisierung üblich, die Anpassung der Gastsysteme.

Ohne dedizierte Hardware-Unterstützung und ohne aufwendige Anpassungen der Gastsysteme kommt die Virtualisierungslösung in [59] aus. Die Autoren nutzen die kleinen lokalen Speicher, welche auf einigen eingebetteten Systemen zur exklusiven Nutzung durch die jeweiligen Rechenkerne zur Verfügung stehen, um lokale Hypervisorinstanzen von den Gastbetriebssystemen zu separieren und diese dennoch gemeinsam im Kernelbereich auszuführen. Diese lokalen Speicher liegen unter

vollständiger Kontrolle der lokalen Software, so dass deren Inhalt, anders als bei Caches, nicht durch automatische Ersetzungsstrategien in den Hauptspeicher gelangt, wo er eventuell durch andere Anwendungen manipuliert werden kann.

Verteilte Instanzen des vorgestellten Hypervisors SPUMONE liegen physisch getrennt in den lokalen Speichern, welche von anderen Rechenkernen aus nicht erreichbar sind, um so eine Separierung von unterschiedlich kritischen Anwendungen herstellen zu können. Anders als ein zentraler Hypervisor oder Microkernel können die Instanzen unabhängig und ohne Fortpflanzung von eventuellen Fehlern in der Virtualisierungsschicht ausfallen. Ein weiterer Vorteil einer verteilte Virtualisierungslösung ist die bessere Skalierbarkeit für große Systeme mit vielen Rechenkernen.

Von den lokalen virtuellen Maschinen, welche weiterhin im Hauptspeicher laufen, kann auf den lokalen Speicherbereich nur nach expliziter Autorisierung durch den lokalen Hypervisor zugegriffen werden. Durch die physische Trennung können die Betriebssystemkerne des Gastsystems mit nur wenig Software-Änderungen und Emulationsaufwand weiterhin zusammen mit der lokalen Hypervisorinstanz sicher im Kernelbereich ausgeführt werden.

Die Autoren von [71] verwenden eine Kombination aus klassischen Hypervisoren auf den einzelnen Kacheln einer Vielkernplattform und einer Hardware-Erweiterung für die Virtualisierung der Kommunikation mit verteilten Ressourcen. Einen Hypervisor, der sich über mehrere Kacheln erstreckt, halten die Autoren, wie in [59] auch, für ein zu großes Risiko einer einzelnen Schwachstelle für den Ausfall des gesamten Systems (engl. *Single Point of Failure*). Weiterhin steigt der Aufwand, um den Speicher für Anwendungen, die sich über mehrere Kacheln erstrecken, kohärent zu halten. Sowohl Hypervisor als auch Anwendungen sind daher in der vorgestellten Arbeit auf einzelne Kacheln begrenzt und kommunizieren über dedizierte Nachrichten. Die Kommunikation mit anderen Komponenten auf dem Chip wird als besonders zeitkritisch betrachtet, weshalb hier eine Hardware-Unterstützung, welche auf einer virtuellen Netzwerkkarte [70] basiert, für die Virtualisierung der Kommunikationsverbindungen eingesetzt wird.

Die Virtualisierung wird in der vorgestellten Arbeit für die transparente Migration von Aufgaben von fehlerhaften Kacheln auf Reservekacheln eingesetzt. Hierfür wird, abhängig von den Kosten der Migration, eine Grenze festgelegt. Auftretende Fehler einer Aufgabe werden nach der Kritikalität und der Empfindlichkeit der Aufgabe gegenüber Fehlern gewichtet. Übersteigt die gewichtete Summe die zuvor definierte Grenze, wird die betroffenen Aufgabe auf eine andere Kachel migriert. Abhängig von der Kritikalität einer Aufgabe wird die Migrationsstrategie und damit die Geschwindigkeit der Migration festgelegt. Zur Auswahl stehen heiße Migration, wobei die Aufgabe an der alternativen Position bereits läuft, warme Migration mit einer bereits vorkonfigurierten jedoch nicht laufenden Alternative und kalte Migration auf eine nicht vorbereitete Kachel.

2.5. Kommunikation

Die Kommunikation zwischen den einzelnen Rechenkernen einer Mehrkernplattform kann, wie bereits in [1] dargestellt, über Speicherbereiche, die sowohl für die Sender als auch für die Empfänger einer Nachricht zugreifbar sind, oder, wie beispielsweise bei Intels SCC, über dedizierte Puffer für den Nachrichtenaustausch realisiert werden. Die Kommunikation zwischen Rechenkernen und Peripheriemodulen geschieht über das Setzen von Statusbits, um so ein neues Ereignis für einen oder mehrere Rechenkerne anzuzeigen.

Anwendungen können über das periodische Lesen der Nachrichtenpuffer [74] sowie der Statusbits aller vorhandenen Peripheriemodule Kenntnis über neue Nachrichten oder Ereignisse erlangen. Dieser Ansatz leidet jedoch unter einer möglicherweise langen Verzögerung durch das Lesen aller möglichen Nachrichtenpuffer und Statusbits. Darüber hinaus würde die Zeit, um auf eine neue Nachricht eines anderen Rechenkerns oder ein neues Ereignis eines Peripheriemoduls zu reagieren, mit der Größe des Systems, das heißt, mit der Anzahl an Nachrichtenpuffern und Peripheriemodulen, oder auch mit der Periode des Überprüfens wachsen. Daher werden neue Nachrichten und Ereignisse in den meisten Mikroprozessoren effizient über Interrupt-Anfragen bei den entsprechenden Rechenkernen angekündigt. Interrupts können mehreren Zielen zugeordnet werden, um Komponenten gemeinsam zu nutzen oder um Aufgaben redundant auszuführen. Die Änderung der Zuordnungen von Interrupt-Quellen zu Interrupt-Zielen lässt sich auch zur Laufzeit relativ einfach über das Maskieren und Demaskieren von Interrupts realisieren.

Um für eine Vielkernplattform mit möglicherweise unterschiedlich kritischen Anwendungen geeignet zu sein, muss ein Interrupt-Mechanismus jedoch weitere Bedingungen erfüllen:

- Er muss skalierbar sein, um auch eine sehr große Anzahl an Interrupt-Quellen und -Zielen effizient unterstützen zu können.
- Er muss transparent und flexibel sein, um die Migration von Anwendungen auf andere Kacheln sowie das Austauschen von Peripheriemodulen zur Laufzeit zu unterstützen und das am besten ohne die beteiligte Software anpassen zu müssen.
- Er muss die sichere Trennung der Interrupt-Behandlung von unterschiedlich kritischen Anwendungen erlauben.

Indem virtuelle anstatt physische Interrupt-Quellen und -Ziele verwendet werden und die tatsächliche Zuordnung einer dritten unabhängigen Instanz überlassen wird, können kommunizierende Anwendungen ohne Software-Anpassungen transparent und flexibel auf beliebigen Kacheln platziert werden und beliebige verteilte Komponenten verwenden. Dies ist besonders entscheidend, wenn Anwendungen zur Laufzeit auf andere Kacheln verschoben oder fehlerhafte Peripheriemodule ersetzt werden sollen. Rauchfuss et. al [71] ermöglichen auf ihrer Plattform zwar eine Form der Virtualisierung und unterstützen eine transparenten Migration von Aufgaben, deren Virtualisierung von Kommunikationsverbindungen zielt jedoch auf die gemeinsame Verwendung von Netzwerkschnittstellen durch mehrere virtuelle Maschinen und nicht auf eine transparente und flexible Übersetzung zwischen virtuellen und tatsächlich vorhandenen physischen Rechenkernen und Peripheriemodulen.

Intels SCC bietet neben der periodischen Überprüfung der Nachrichtenpuffer zwar auch die Möglichkeit, explizite Interrupt-Anfragen zwischen den Rechenkernen zu versenden [50], dafür muss der physische Zielrechenkern jedoch der versendenden Anwendung bekannt sein, wodurch eine transparente und flexible Migration von Anwendungen zwischen Rechenkernen ausgeschlossen ist. Alternativ bietet Intels SCC zusätzlich eine globale Interrupt-Steuerung an, welche in einem gewissen Rahmen eine transparente Migration von kommunizierenden Aufgaben erlauben würde. Mit einer zentralen Interrupt-Steuerung könnte ein Interrupt für den ursprünglichen Zielrechenkern vor der Migration maskiert und für den neuen Zielrechenkern aktiviert werden, ohne dass die migrierte Aufgabe dafür angepasst werden müsste.

Eine weitere zentrale Interrupt-Steuerung, welche die Zuordnung von Peripheriemodulen zu individuellen Rechenkernen sowie das Senden von Interrupt-Anfragen an mehrere (engl. *Multicast*) oder auch alle (engl. *Broadcast*) anderen Rechenkerne eines Mehrprozessorsystems erlaubt, wird in [85] beschrieben. Diese und alle anderen Lösungen mit einer zentralen Interrupt-Steuerung sind jedoch nicht skalierbar, um für eine möglicherweise sehr große Anzahl an Interrupt-Quellen und -Zielen einer Vielkernplattform einsetzbar zu sein. Weiterhin erlaubt eine zentrale Interrupt-Steuerung kein *Asymmetric Multiprocessing (AMP)* mit mehreren unabhängigen und unterschiedlich kritischen Anwendungen beziehungsweise Betriebssystemen, da Zugriffe von verschiedenen Anwendungen und Betriebssystemen auf eine zentrale Einheit nicht oder nur mit großem Aufwand sicher separiert werden können.

Eine Lösung hierfür könnte eine zusätzliche schlanke Software-Schicht zwischen den Anwendungen beziehungsweise Betriebssystemen und der eigentlichen Interrupt-Steuerung in Hardware sein, wie ein *Microvisor* [44], *Microhypervisor* [81] oder ähnliche Software-Virtualisierungslösungen, wie in Abschnitt 2.4 erläutert wird. Diese Schicht von höchster Kritikalität könnte Zugriffe auf eine zentrale oder auch verteilte Interrupt-Steuerung und andere gemeinsam verwendete Komponenten kontrollieren und sicher voneinander trennen. Alle Virtualisierungslösungen benötigen jedoch, wie in Abschnitt 2.4 beschrieben, entweder eine Hardware-Unterstützung für eine vollständige Virtualisierung [81], welche jedoch in den meisten eingebetteten Systemen nicht vorhanden ist, oder eine Reihe von Modifikationen des Gastsystems für eine Paravirtualisierung [44], was vor allem die Integration von bereits existierenden Anwendungen erschwert.

Weiterhin können Virtualisierungslösungen mit einem einzelnen Systemkern für große Systeme zu einem Engpass werden, wenn durch die große Menge an möglichen Interrupt-Anfragen auf einer Vielkernplattform sehr viele Kontextwechsel erforderlich werden. Darüber hinaus unterstützen die meisten existierenden Virtualisierungslösungen keine heterogenen Systeme mit unterschiedlichen Rechenkernen.

Als Ausnahmen sind hier Barrelfish [13] und RTEMS [65] zu nennen. RTEMS wird zwar speziell für Mehrprozessorsysteme entwickelt, unterstützt aber weder eine dynamische Migration von Aufgaben auf andere Rechenkerne noch das Ersetzen von fehlerhaften Komponenten und ist daher nur bedingt für sicherheitskritische Mehrprozessorsysteme einsetzbar. Die Kommunikation zwischen Rechenkernen wird bei RTEMS über eine separate Kommunikationsschicht realisiert. Der *Multiprocessor Communications Interface Layer (MPCI)* ist vom Anwender selbst zu implementieren und erzeugt typischerweise Interrupt-Anfragen nach Empfang einer neuen Nachricht, kann hierbei jedoch normalerweise nicht auf Abweichungen vom spezifizierten Verhalten reagieren. Da Barrelfish ebenfalls unabhängig von der tatsächlichen Implementierung eines Kommunikationsmechanismus zwischen Rechenkernen ist, könnten sowohl Barrelfish als auch RTEMS von einem sicheren und flexiblen Interrupt-Mechanismus profitieren, wie er in den Abschnitten 4.2.9 und 4.2.10 beschrieben wird.

2.6. Plattformen

Dem Trend zu mehr aber dafür meist einfacheren Rechenkernen auf einem einzelnen Chip folgend, wurden in den letzten Jahren eine Vielzahl von Mehr- und Vielkernplattformen vorgestellt. Im Folgenden wird hiervon eine Auswahl mit Bezug zu der vorliegenden Arbeit vorgestellt.

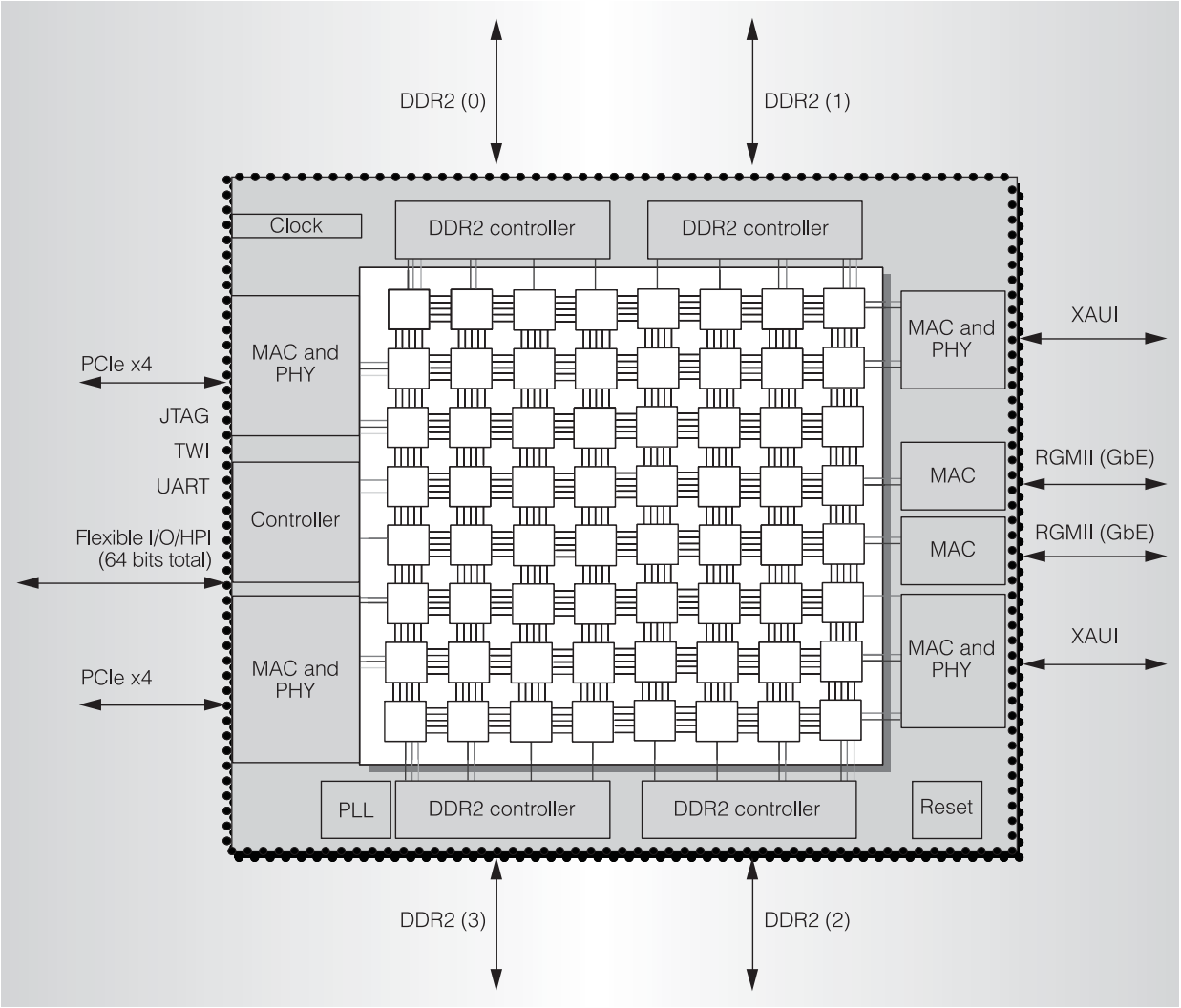

Abbildung 2.5.: Tileras TILE64 [91]

2.6.1. Tileras TILE64

Der TILE64-Prozessor [91] ist, wie in Abbildung 2.5 dargestellt, aus 64 identischen Kacheln (engl. *Tiles*) in einem Acht-mal-Acht-Array als MIMD-Maschine aufgebaut. An den Rändern des Chips liegen I/O- und Speicherschnittstellen, wie PCIe und DDR2. Die Kacheln bestehen jeweils aus einem universellen Rechenelement, Cache und einem Switch. Jede Kachel führt entweder eine Anwendung direkt auf der Hardware, eine komplettes unabhängiges Betriebssystem oder zusammen mit einer Gruppe von Kacheln ein gemeinsames Betriebssystem als SMP-System aus. Jede Kachel verfügt über eine separate Steuerung für *Direct Memory Access (DMA)*, um Daten zwischen Speicher und Kachel ohne Einsatz des Rechenelements zu transportieren, eine eigene Interrupt-Steuerung sowie eine Hardware-Unterstützung für Cache-Kohärenz. Der virtuelle lokale Adressraum von 32 Bits wird über separate TLBs in den Kacheln in den 36-Bit großen globalen physischen Adressraum übersetzt [14]. Die Anwendungen auf den Kacheln kommunizieren über gemeinsam verwendete Speicherbereiche oder über ein dediziertes Kommunikationsnetzwerk.

Die Elemente des Arrays werden über fünf separate Netzwerke mit unterschiedlichen dedizierten Einsatzbereichen verbunden. Vier der Netzwerke sind dynamische Netzwerke, die ein paketbasiertes *wormhole-switched* Verfahren und X/Y-Routing einsetzen. Dies bedeutet, die Teile eines Paketes wandern ähnlich wie ein Wurm von den Puffern eines Switches zu den Puffern des nächsten Switches und bewegen sich so von der Quelle zur Senke. Dabei bewegen sich die Teilpakete zunächst in X-Richtung bis zur korrekten Spalte des Arrays und dann in Y-Richtung bis die Zielkachel erreicht ist. Die dynamischen Netzwerke werden für Speicheranfragen, Speicherdatenverkehr, Nachrichten zwischen Anwendungen und für Systemnachrichten verwendet. Das fünfte Netzwerk ist ein statisches Netzwerk mit einer beim Start der Anwendung einmalig festgelegten Route. Das statische Netzwerk kann für Daten-Streaming zwischen parallelen Anwendungen eingesetzt werden.

In [60] wird der TILE64 als Basis für einen Vielkernprozessor für Raumfahrtanwendungen eingesetzt. Auf dem vorgestellten Maestro-Chip kommen 49 Kacheln in einem Sieben-mal-Sieben-Array zum Einsatz. Die Rechenkerne des TILE64 wurden dabei um eine Fließkommaeinheit erweitert. Um die Empfindlichkeit gegenüber *Single Event Upsets (SEUs)* für den Einsatz im Weltraum zu reduzieren, wurde eine spezielle weniger strahlungsempfindliche ASIC-Bibliothek eingesetzt. SEUs sind Zustandsänderungen eines Logikelements durch Einschläge ionisierender Partikel. Weiterhin wurden dem System spezielle I/O-Schnittstellen für Raumfahrtanwendungen hinzugefügt.

Villapando et al. [89] untersuchten systematisch, was über den Einsatz von speziellen ASIC-Bibliotheken hinaus für Änderungen an dem TILE64 nötig wären, um ihn, auch unter Beachtung wirtschaftlicher Aspekte, zuverlässig für Raumfahrtanwendungen einsetzbar zu machen. Der Meastro-Chip mit seiner reduzierten Anfälligkeit für SEUs aber ohne eine grundlegende Fehlertoleranz wurde als Startpunkt definiert. Um die notwendigen Änderungen so gering wie möglich zu halten, wird in der genannten Arbeit eine Software-Lösung vorgeschlagen. Die Systemsteuerung soll dafür dreifach redundant auf drei Kachel mit einer zusätzlichen externe Überwachungsschaltung realisiert werden. Die Anwendungen selbst sollen entweder mit einem Selbsttest ausgestattet werden oder ebenfalls mehrfach redundant auf mehreren Kacheln ausgeführt werden.

Für die Implementierung eines mehrfach redundanten Systems mit Entscheidungseinheit (engl. *Voter*) auf einer Kachel-Architektur ist es nötig, dass Fehler auf einzelne Instanzen begrenzt bleiben und dass eine zuverlässige Kommunikation sichergestellt ist [89]. Für eine zuverlässige Kommunikation muss für den Empfänger erkennbar sein, ob eine Nachricht verfälscht wurde und ob sie von dem korrekten Sender abgeschickt wurde. Hierfür schlagen die Autoren einen Fehlererkennungsmechanismus, wie beispielsweise zusätzliche Paritätsbits, für die entsprechenden Netzwerke sowie ein Hinzufügen der Identifikationsnummer des Absenders zu allen Paketen vor. Um fehlerhafte Switches umgehen und mehrere redundante Routen verwenden zu können, schlagen die Autoren weiterhin eine Anpassung des verwendeten Routing-Verfahrens vor.

Damit Fehler auf einzelne Instanzen einer redundant ausgeführten Anwendung begrenzt bleiben, werden in [89] vor allem Mechanismen vorgeschlagen, um den virtuellen Adressraum einer Instanz durch fehlerhafte Zugriff von anderen Instanzen zu schützen. Die Inhalte des Speichers und Caches auf dem TILE64 sind bereits mit Codes zur Fehlererkennung und Fehlerkorrektur versehen, die TLBs und DMA-Einheiten jedoch nicht, was zu einer Verletzung der Grenzen des virtuellen Adressraumes einzelner Anwendungsinstanzen führen kann. Es wird daher empfohlen, die Codierung zur Fehlerkorrektur auf TLBs und DMA-Einheiten auszuweiten. Weiterhin wird empfohlen, die *Multicore Hardwall*, welche für drei der fünf Netzwerke des TILE64 zur Aufteilung des Systems in

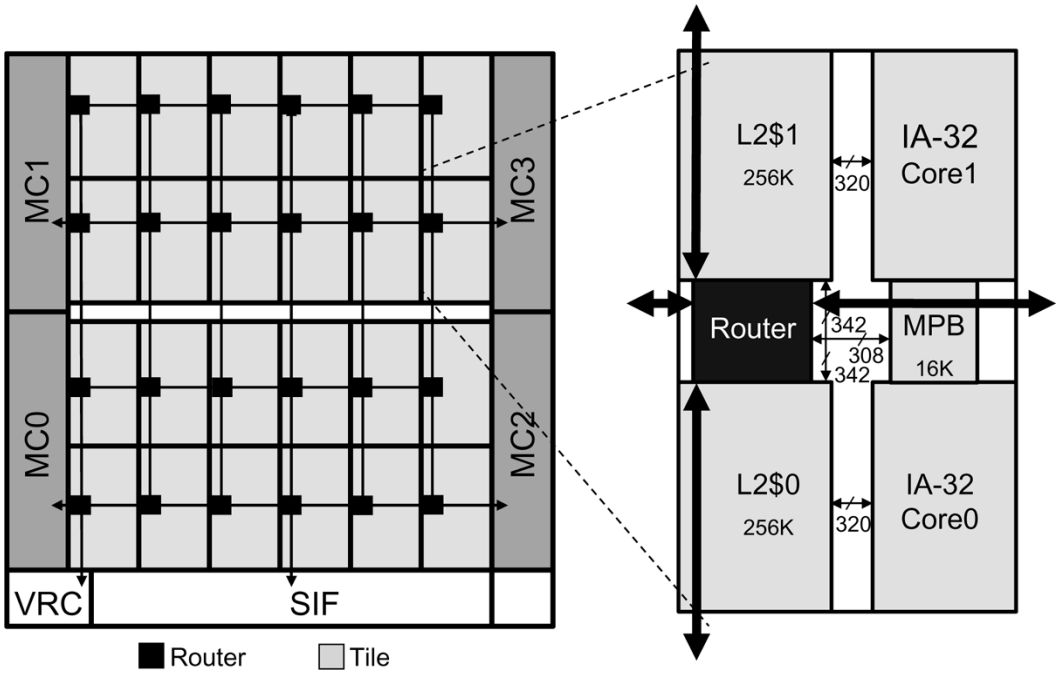

Abbildung 2.6.: Intels Single-Chip Cloud Computer (SCC) [47]

einzelne Regionen existiert und in Abschnitt 2.2.1 genauer erläutert wird, auch für Schreibzugriffe auf das Netzwerk für Speicherdatenverkehr zu erweitern, um die Folgen von Hardware-Fehlern in den TLBs oder Netzwerken eingrenzen zu können.

Zusätzlich schlagen die Autoren Schutzmaßnahmen für Ressourcen mit Einfluss auf den globalen Systemzustand, wie Rechenelemente, die eine Hypervisorinstanz ausführen, die Taktversorgung oder die Gesamtsystemsteuerung vor.

2.6.2. Intels Single-Chip Cloud Computer (SCC)

Intels Single-Chip Cloud Computer [47] ähnelt Tileras TILE64 im grundsätzlichen Aufbau, unterscheidet sich jedoch in einigen Implementierungsdetails. Der SCC besteht ebenfalls aus Kacheln, die in einem zweidimensionalen Array angeordnet sind. An den Rändern des Arrays befinden sich vier DDR3-Schnittstellen sowie eine Hochgeschwindigkeits-I/O-Schnittstelle. Die 24 Kacheln des SCCs sind über ein Sechs-mal-Vier-Netzwerk verbunden. Eine Kachel beinhaltet, wie in Abbildung 2.6 dargestellt, jeweils zwei Rechenkerne mit individuellem L1- und L2-Cache mit integrierter Fehlerkorrektur, einer gemeinsamen Netzwerkschnittstelle sowie einem geteilten lokalen Speicher für den schnellen Austausch von kurzen Nachrichten, welcher *Message Passing Buffer (MPB)* genannt wird. Für Nachrichten, die mehr als 16 Kilobytes umfassen, muss auf externe Speicher ausgewichen werden. Eine Hardware-Unterstützung für Cache-Kohärenz von gemeinsam verwendeten Speicherbereichen ist auf dem System nicht vorhanden. Falls Cache-Kohärenz benötigt wird, kann diese jedoch über Software realisiert werden.

Die Umsetzung des vier Gigabytes großen Adressraumes der verwendeten Rechenkerne in den 64 Gigabytes großen Adressraum des Gesamtsystems geschieht über Tabellen (engl. *Lookup Tables (LUTs)*), welche auch die Routen zu den entsprechenden Zielen im Netzwerk enthalten. Die

LUTs sind dynamisch konfigurierbar, auch von anderen Rechenkernen, was eine Separierung von unterschiedlichen Anwendungen erschwert.

Beim SCC wird ebenfalls X/Y-Routing eingesetzt, die Pakete enthalten jedoch, anders als beim TILE64 von Tilera, die Identifikationsnummer der Paketquelle. Die Router des Netzwerkes verfügen über weitere vier Ports, um eine Kachel in Nord-, Ost-, Süd- und Westrichtung mit benachbarten Kacheln zu verbinden. Der SCC verwendet das *Virtual-Cut-Through-Switching*-Verfahren, um die Verzögerung gegenüber einem *Wormhole-Switching*-Verfahren, wie beim TILE64 eingesetzt, zu reduzieren. Beim *Virtual-Cut-Through-Switching*-Verfahren wird Pufferplatz für ganze Pakete reserviert und nicht nur für Teilpaketen (engl. *Flow Control Digits (Flits)*) wie beim *Wormhole-Switching*-Verfahren. Um Blockierungen zu vermeiden, bietet der SCC acht virtuelle Kanäle, wovon sechs frei für Anwendungen verwendbar sind und damit auch für die Separierung von Paketen von unterschiedlich kritischen Anwendungen eingesetzt werden könnten. Die Zuordnung von Paketen zu virtuellen Kanälen kann jedoch auch spekulativ wechseln, um eine gleichmäßige Auslastung der Kanäle zu erhalten. Das Netzwerk verfügt darüber hinaus über zusätzliche Signale für die Fehlerkorrektur [11].

Das feingranulare Energiemanagement über die zentrale Einheit *Voltage-Regulator-Control (VRC)* des SCC bietet einstellbare Spannungen und Frequenzen für acht beziehungsweise 28 unterschiedliche Regionen. Das zentrale Energiemanagement ist jedoch, ebenfalls wie die LUTs der einzelnen Rechenkerne für die Adressumsetzung, von allen Rechenkernen aus erreichbar. Eine Realisierung der Separierung von unterschiedlich kritischen Anwendungen würde sich daher auf dem SCC ohne eine zusätzliche Virtualisierungsschicht kaum realisieren lassen.

2.6.3. Next Generation Multipurpose Microprocessor (NGMP)

Der Next Generation Multipurpose Microprocessor (NGMP) ist ein weiterer Mehrkernprozessor für Raumfahrtanwendungen [6]. Daher steht auch hier im Vordergrund, die Empfindlichkeit gegenüber SEUs zu reduzieren, auftretende SEUs zu erkennen und zu korrigieren. Beim NGMP sind dafür sämtliche Speicher, wie interne Registerbänke und L1-Caches der eingesetzten Prozessoren, der geteilte L2-Cache, externe Speicher wie DDR2, SDRAM und PROM, sowie interne Speicher aller verwendeten Peripheriemodule über *Error Correction Codes (ECCs)*, Paritätsbits oder TMR geschützt. Darüber hinaus sind auch alle Flipflops aus einer speziellen für SEUs weniger anfälligen Bibliothek implementiert und, wo dies nicht möglich ist, sind die Flipflops dreifach redundant mit Entscheider ausgelegt. Transiente Fehler im DDR2-Speicher und dem SDRAM können über eine *Memory-Scrubbing*-Einheit und ECC korrigiert werden, bevor nicht mehr korrigierbare Mehrbitfehler auftreten. Beim *Memory-Scrubbing* werden mit ECC geschützte Speicher Eintrag für Eintrag ausgelesen, gegebenenfalls korrigiert und wieder zurückgeschrieben. Permanente Fehler in einzelnen Blöcken des DDR2-Speichers können bis zu einem gewissen Grad über Reserveblöcke behoben werden.

Der NGMP basiert auf dem aktuellen LEON-Prozessor von Aeroflex Gaisler in der fehlertoleranten Ausführen, dem LEON4FT. Die vier vorhandenen Prozessoren beinhalten, wie in Abbildung 2.7 dargestellt, jeweils eine eigene MMU und einen exklusiv genutzten L1-Cache. Der L2-Cache wird von allen Prozessoren gemeinsam verwendet. Teile des L2-Caches können auch als fehlertoleranter lokaler Speicher eingesetzt werden. Die Prozessoren können, wenn sie als SMP-System mit einem gemeinsamen Betriebssystem auf allen Rechenkernen eingesetzt werden, eine gemeinsame Interrupt-Steuerung und gemeinsame Zeitgeber verwenden oder, als AMP-System mit separaten Anwendungen

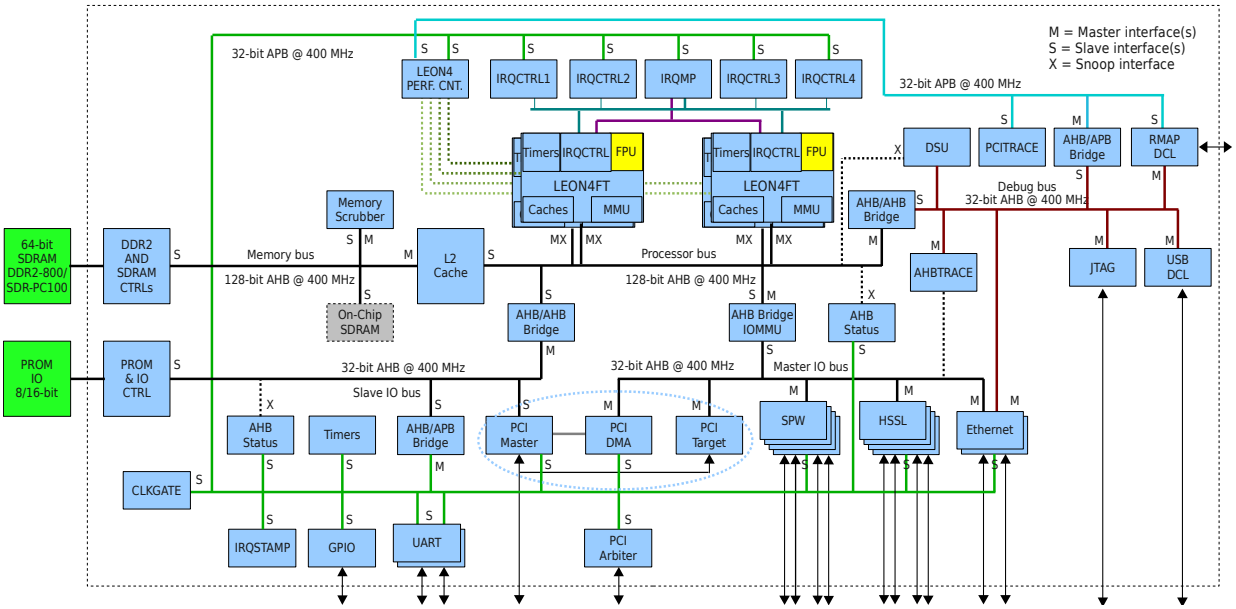

Abbildung 2.7.: Next Generation Multipurpose Microprocessor [6]

oder Betriebssystemen auf den einzelnen Prozessoren, separate Interrupt-Steuerungen und Zeitgeber verwenden.

Bei einem AMP-System können die verfügbaren MMUs für die räumliche Trennung der Anwendungen beziehungsweise Betriebssysteme auf den einzelnen Prozessoren eingesetzt werden, wobei diese sich dann, wie in Abschnitt 2.2 beschrieben, vertrauen können müssen, also demselben Kritikalitätslevel entsprechen müssen. Dies gilt jedenfalls für die Software-Elemente, welche für die Konfiguration der jeweiligen MMU zuständig sind, also beispielsweise den Hypervisor oder das Betriebssystem. Darüberliegende Aufgaben können dann wiederum auch unterschiedlich strengen Sicherheitsanforderungen genügen, wenn Veränderungen der MMU-Konfiguration durch diese Aufgaben ausgeschlossen werden können.

Die Prozessoren des NGMP sind alle mit dem 128-Bit breiten Prozessorbus verbunden. Daneben existieren noch vier weitere separate Busse, ein Debug-Bus, ein Speicherbus und zwei Peripheriebusse, einmal für Slaves und einmal für Master. Der Master-Peripheriebus ist über eine *Input/Output Memory Management Unit (IOMMU)* mit dem Prozessorbus verbunden, um auch hier eine Separierung gegenüber Peripheriemodulen, die aktiv auf das restliche System zugreifen können, sicherstellen zu können.

Fernandez et al. untersuchen in [35] die zeitliche Separierung beziehungsweise die maximal mögliche gegenseitige Beeinflussung von unterschiedlichen Aufgaben auf dem NGMP. Die Autoren verwenden Mikrobenchmarks, um gemeinsam verwendete Teile des Systems besonders zu belasten, um dann eine obere Grenze für die Beeinflussung auf parallel ausgeführte Aufgaben zu ermitteln. Um so höher die Unterschiede der Antwortzeiten von parallel ausgeführten Aufgaben durch die Belastung durch die Mikrobenchmarks sind, desto weniger ist eine Architektur, nach Auffassung der Autoren, für Echtzeitanwendungen geeignet. Als wichtigste Quellen für eine gegenseitig Beeinflussung identifizierten die Autoren den Prozessorbus, den L2-Cache, die Speichersteuerung sowie die Cache-Strategie für den L1-Cache. Dementsprechend werden in den verwendeten Mikrobenchmarks

vor allem Operation verwendet, welche Aktivität in den genannten Systemteilen hervorrufen. Vor allem Schreibzugriffe auf den L1-Cache haben mit der auf dem NGMP eingesetzten Cache-Strategie in den präsentierten Experimenten einen großen Einfluss auf andere Aufgaben, da hierbei jedes Mal auch auf den gemeinsam verwendeten Prozessorbus zugegriffen wird. Die Autoren schlagen daher unter anderem eine Änderung der Strategie des L1-Caches vor, um das Zeitverhalten von unterschiedlichen Anwendungen auf dem NGMP unabhängiger zu machen.

2.6.4. ACROSS MPSoC

Das ACROSS Multiprocessor System-on-Chip (MPSoC) ist ein konfigurierbares System aus heterogenen Komponenten, die, wie in Abbildung 2.8 dargestellt, über ein NoC verbunden sind. Bei der Entwicklung stand die Reduzierung der Komplexität im Vordergrund, um selbst hoch kritische und nicht kritische Anwendungen gemeinsam auf einer eingebetteten Plattform integrieren zu können. Das System soll eine unabhängige Entwicklung und Verifikation der einzelnen Teilsysteme erlauben. Hierfür wurden die Teilsysteme des MPSoCs wie einzelne vernetzte aber zeitlich und räumlich getrennte Systeme mit eigenem lokalen Speicher betrachtet. Eine Interaktion zwischen den Teilsystemen ist nur über dedizierte Schnittstellen, die TISSes (*Trusted Interface Subsystem*), mittels Nachrichten von einer definierten Länge und zu vorher definierten Zeitpunkten möglich. Die TISSes verhindern eine ungewollte oder fehlerhafte gegenseitige Beeinflussung der einzelnen Subsysteme und werden auch für die Konfiguration der einzelnen Komponenten verwendet. Die Subsysteme werden so zu fehlereingrenzenden Regionen für alle systematischen Fehler. Für zufällige Fehler, wie SEUs oder Hardware-Defekte in gemeinsam verwendeten Ressourcen, wie beispielsweise der Spannungsversorgung, stellt das gesamte System eine fehlereingrenzende Region dar und erfordert weitere externe Maßnahmen, die jedoch nicht Bestandteil der in [75] vorgestellten Arbeit sind.

Der Kern des ACROSS MPSoC ist das eingesetzte zeitgesteuerte (engl. *time-triggered*) NoC, welches aus einem vermaschten Netz aus Switches mit jeweils vier Schnittstellen besteht. Diese können entweder zum Anschließen von Komponenten oder von anderen Switches verwendet werden. Die Weiterleitung von Daten geschieht auch hier paketbasiert und unter Verwendung des *Wormhole-Switching*-Verfahrens. Das NoC bietet unabhängige Kanäle auf dem geteilten physischen Medium, um neben der zeitlichen Separierung auch eine räumliche Separierung von unterschiedlichen Nachrichten im NoC sicherzustellen. Durch die räumliche Trennung wird verhindert, dass Nachrichten von einer Komponente durch Nachrichten von einer anderen Komponente überschrieben oder verändert werden. Die Switches können gleichzeitig Pakete in unterschiedliche Richtungen weiterleiten. Damit zur gleichen Zeit jedoch nie mehr als ein Paket an denselben Ausgangsport weitergeleitet werden soll, wird die zeitliche Separierung vorher durch einen festgelegten zeitlichen Ablaufplan sichergestellt. Durch den festen Zeitplan wird eine deterministische Interaktion zwischen den Komponenten ohne Kollisionen von Nachrichten sichergestellt, was eine Arbitrierung zur Laufzeit und damit eine aufwendige Analyse des Zeitverhaltens überflüssig macht.

Der zur Entwurfszeit festgelegte Zeitplan legt fest, zu welchen Zeitpunkten eine Komponente eine Nachricht senden beziehungsweise empfangen kann. Hierfür wird eine systemweit einheitliche Zeitbasis eingeführt. Die Zeitpläne sind in den TISSes abgelegt, können von den Anwendungen auf den lokalen Komponenten jedoch nicht verändert werden. Eine Änderung zur Laufzeit ist nur zu vorher festgelegten Zeitpunkten durch eine zentrale Instanz, den *Trusted-Resource-Manager (TRM)*, möglich.

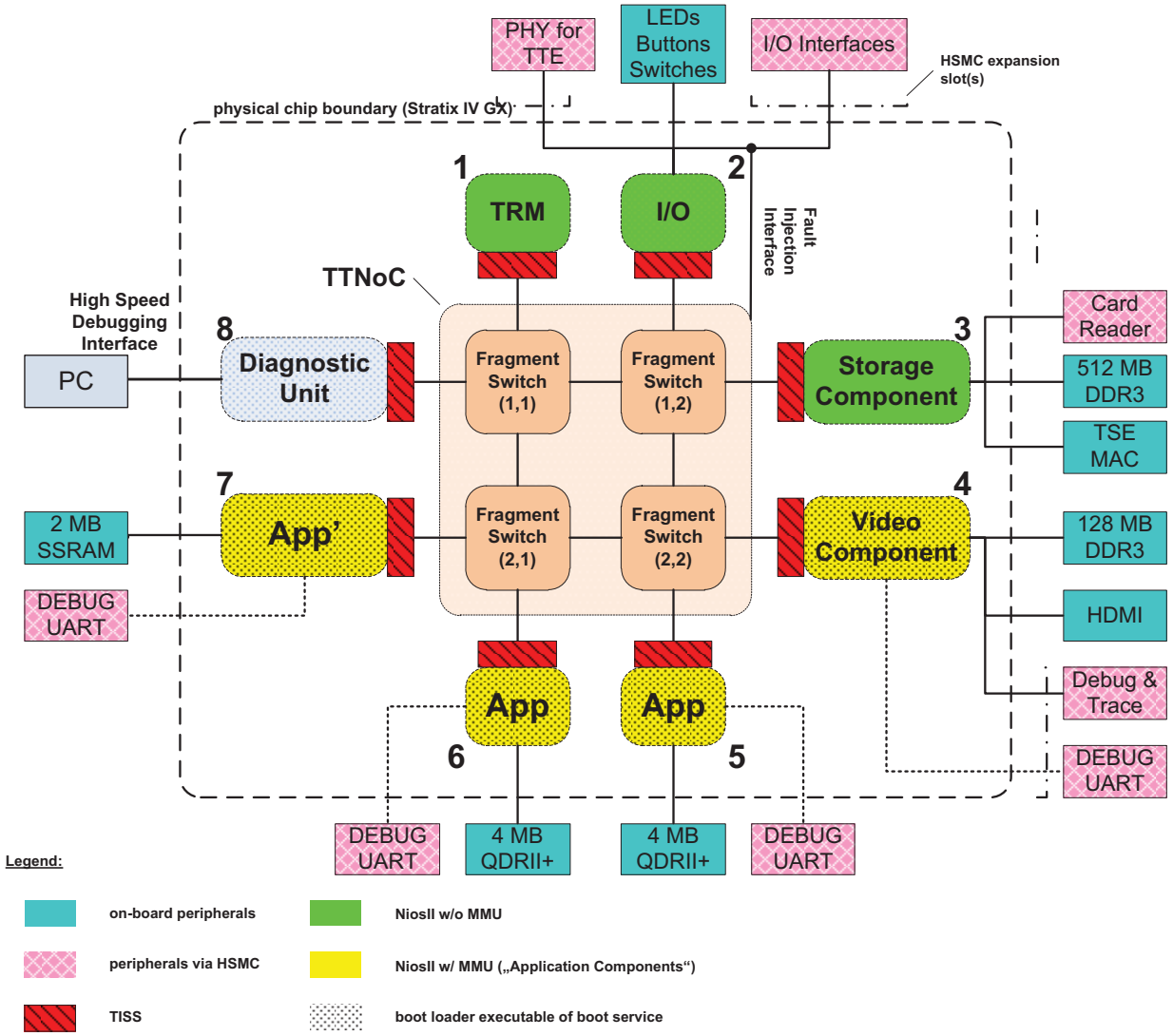

Abbildung 2.8.: ACROSS MPSoC [75]

Fehlerhafte Komponenten können über eine lokale Fehlererkennung in den TISSes zurückgesetzt werden. Alternativ kann die lokal ausgeführte Anwendung durch einen Interrupt benachrichtigt werden. Die Fehlererkennung kann für lokale Anwendungen als ausschließlich lesbar konfiguriert werden, um so eine Maskierung von Fehlern durch eine lokale Anwendung zu verhindern. In diesem Fall sammelt ein zentraler Service alle erkannten Fehler ein, sendet diese an eine Diagnoseeinheit und kann anschließend auch die Fehlermeldung in den Komponenten zurücksetzten.

Abbildung 2.8 zeigt ein Beispielsystem des ACROSS MPSoC mit acht Komponenten, welche über vier Switches verbunden sind. Die TISSes für die Separierung und Konfiguration der einzelnen Komponenten sind in der Abbildung in rot dargestellt. Vier der Komponenten sind als Systemkomponenten konfiguriert und realisieren zentrale Funktionen, wie das Aufzeichnen von Fehlern und anderen Ereignissen, das Debugging der Plattform sowie den oben genannten *Trusted-Resource-Manager*. Weiterhin bieten sie einen zentralen Zugriff auf I/O-Schnittstellen und externe Speicher. Die übrigen vier Komponenten werden für Anwendungen eingesetzt, welche in der vorgestellten

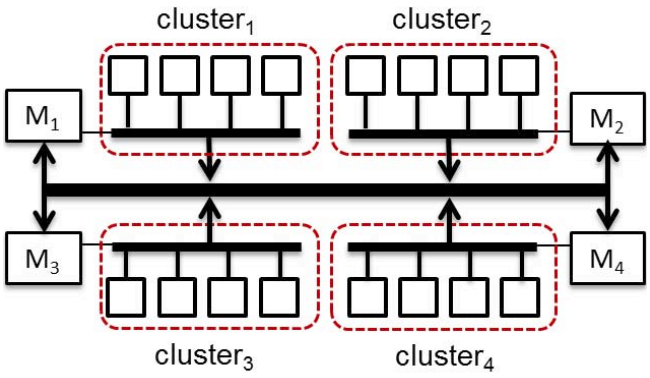

Abbildung 2.9.: parMERASA-Architektur [86]

Arbeit auf NIOS-II-Prozessoren von Altera und dem Betriebssystem PikeOS von Sysgo ausgeführt werden.

2.6.5. Weitere Plattformen

Das Projekt parMERASA [86] befasst sich, über die Integration von mehreren unterschiedlich kritischen Anwendungen hinaus, mit der Parallelisierung einzelner sequentieller Echtzeitanwendungen mit besonders hohen Anforderungen an die Rechenleistung. Hierfür werden, neben einer Rechenplattform, Methoden für eine analysierbare Parallelisierung sowie für die Analyse geeignete Software-Programme entwickelt.

Die skalierbare Rechenplattform mit bis zu 64 Rechenkernen bietet eine zeitliche und räumliche Separierung unterschiedlich kritischer Anwendungen, zum einen über echtzeitfähige Betriebssystemkerne und zum anderen über eine Architektur mit Gruppen von Rechenkernen, die über separaten NoCs verbunden sind, wie in Abbildung 2.9 dargestellt. Unterschiedliche Anwendungen werden auf jeweils unterschiedlichen Gruppen implementiert und können nur über ein weiteres gruppenübergreifendes und analysierbares NoC miteinander kommunizieren. Gruppenübergreifende Kommunikation wird nur sehr eingeschränkt und in einer statischen Weise und damit gut vorhersagbar erwartet.

Das *SmartCore*-System [84] ist eine aus Kacheln aufgebaute Vielkernplattform mit speziellen Routern, um Anwendungen redundant auszuführen, um so deren Zuverlässigkeit zu erhöhen. Das Netzwerk basiert auch hier auf dem *Wormhole-Switching*-Verfahren und X/Y-Routing, bietet jedoch keine virtuellen Kanäle. Die Kacheln selbst gibt es in drei verschiedenen Ausprägungen, als Speicherkacheln, als I/O-Kacheln und als Rechenkacheln. Die Rechenkacheln selbst umfassen einen MIPS-Prozessor, etwas lokalen Speicher sowie eine Netzwerkschnittstelle mit DMA-Steuerung.

Eine Anwendung kann redundant auf mehreren Rechenkacheln ausgeführt werden. Alle hierfür nötigen Schritte werden im Netzwerk vollzogen. Dafür wurden die Router um drei zusätzliche Funktionen erweitert:

- Kopieren, um eingehende Pakete für die redundante Instanz zu duplizieren.
- Ändern der Route, um die Kopie des Paketes nicht an das Originalziel, sondern an die zusätzliche redundante Instanz zu schicken.

- Vergleichen von ausgehenden Paketen mit den ausgehenden Paketen der redundanten Instanz. Bei voneinander abweichenden Paketen kann eine automatische Reaktion ausgelöst werden, wie beispielsweise das erneute Senden einer Nachricht.

Eine weitere auf Kacheln basierende Plattform wird in [54] beschrieben. Die Besonderheit der beschriebenen Plattform mit dem Namen *Heracles* liegt in der quelloffenen und modularen Beschreibung. Die Plattform ist parametrisierbar und vollständig synthetisierbar und erlaubt durch den modularen Aufbau eine Evaluierung unterschiedlicher Designalternativen, wie Anzahl und Art der Rechenkerne, Speicherhierarchien, Speicheraufteilung und Speichergrößen, Routing-Algorithmen, Netzwerkschnittstellen und Netzwerktopologien. Neben einer quelloffenen Implementierung eines MIPS-Prozessors stehen Router mit Unterstützung von virtuellen Kanälen und des *Wormhole-Switching*-Verfahrens zur Verfügung.

2.7. Zusammenfassung

In diesem Kapitel wurden zunächst einige Vorteile von Mehrkernprozessoren aufgeführt sowie Voraussetzungen erläutern, um darauf unterschiedlich kritische Anwendungen sicher, flexibel und transparent auszuführen. Es wurden einzelne Mechanismen und Methoden für die räumliche und zeitliche Separierung sowie Ursachen der gegenseitigen zeitlichen Beeinflussung erläutert. Weiterhin wurden einige Ansätze für die Laufzeitüberwachung, verschiedene Varianten der Virtualisierung sowie Möglichkeiten und Besonderheiten der Kommunikation auf Vielkernplattformen erläutert. Anschließend wurden einige existieren Systeme mit Bezug zu der vorliegenden Arbeit vorgestellt. In Abschnitt 5.4 werden die Ansätze aus diesem Kapitel mit der IDAMC-Plattform und den Mechanismen, welche im Rahmen der vorliegenden Arbeit entwickelt wurden, verglichen.

3 Analyse

Das folgende Kapitel beschreibt, wie das Verhalten von weniger kritischen oder nicht sicherheitsrelevanten Aufgaben bei der Analyse einer kritischen Aufgabe berücksichtigt wird und welchen Einfluss eine Laufzeitüberwachung zur Begrenzung der Beeinflussung haben kann. Der Schwerpunkt liegt hierbei auf der Verwendung von gemeinsamen Ressourcen, wie beispielsweise Speicherschnittstellen, und auf der interrupt-basierten kachelübergreifenden Kommunikation. Auf der Analyse des Zeitverhaltens aufbauend wird eine Analyse des Energieverbrauchs einzelner Anwendungen sowie der Energiedichte in einzelnen Chip-Regionen vorgestellt. Abschließend wird erläutert, welchen Einfluss die Reaktionszeit einer Laufzeitüberwachung auf die Analyse hat.

Auf einer Vielkernplattform werden diverse Ressourcen, wie beispielsweise Speicherschnittstellen, das NoC oder die verfügbare Energie, gemeinsam verwendet. Wenn auf einer Plattform unterschiedlich kritische Anwendungen gemeinsame Ressourcen verwenden, kann eine weniger kritische oder nicht sicherheitsrelevante Anwendung durch eine höhere Fehlerwahrscheinlichkeit höher kritische Anwendungen negativ beeinflussen, zum Beispiel durch die Blockierung einer gemeinsam verwendeten Speicherschnittstelle. Solch eine negative Beeinflussung kann, wie bereits in Kapitel 2 sowie in vorangegangenen Veröffentlichungen [3, 2, 1] erläutert, durch eine ausreichende Separierung von unterschiedlich kritischen Anwendungen verhindert werden.

Eine zeitliche Separierung auf einer Vielkernplattform kann durch eine geeignete Ablaufplanung (engl. *Scheduling*) für alle gemeinsam verwendeten Ressourcen erreicht werden. Ob eine Arbitrierung für eine ausreichende zeitliche Separierung geeignet ist, wird über eine Analyse des Systems zur Entwicklungszeit evaluiert. Baruah et al. erläutern in [12] die Besonderheiten der zeitlichen Ablaufplanung für unterschiedlich kritische Anwendungen auf einem Einzelprozessorsystem. Die zeitliche Ablaufplanung von unterschiedlich kritischen Aufgaben auf einem Mehrkernprozessor wird in [63] beschrieben. Gerade bei höher kritischen Aufgaben liegt die erwartete Antwortzeit für den ungünstigsten Fall, durch eine sehr pessimistische Betrachtung, hierbei meist deutlich über der tatsächlichen Antwortzeit.

Die Einflüsse von unterschiedlich kritischen Anwendungen einer Vielkernplattform auf die Analyse, welche in diesem Kapitel beschrieben werden, müssen alle Möglichkeiten der gegenseitigen Beeinflussung beinhalten. Weicht das Verhalten einer weniger oder nicht kritischen Anwendung, etwa durch einen aufgetretenen Fehler, zur Laufzeit ab, können Garantien, welche hoch kritischen Anwendungen durch die Analyse zur Entwicklungszeit zugesichert wurden, zur Laufzeit nicht mehr eingehalten werden. Es muss also die maximal mögliche Beeinflussung angenommen werden, was gerade bei unterschiedlich kritischen Anwendungen zu einem stark überdimensionierten System führen kann. Umso größer diese Beeinflussung nämlich sein kann, desto weniger Anwendungen können auf einer gemeinsam verwendeten Plattform sicher separiert werden. Es ist also erstrebenswert, besonders die Beeinflussungen von weniger kritischen Anwendungen auf hoch kritische Anwendungen zur Laufzeit durch geeignete Mechanismen oder Methoden so weit wie möglich zu

reduzieren oder zuverlässig vorhersagbar zu machen, um bereits während der Analyse eine geringere Beeinflussung annehmen zu können.

Hierfür könnten weniger oder nicht sicherheitsrelevanten Anwendungen dieselben hohen Anforderungen wie den hoch kritischen Anwendungen auferlegt werden, um deren Verhalten zuverlässig vorhersagen zu können, was jedoch die Gesamtkosten des Systems stark ansteigen lassen würde. Alternativ kann das erwartete Verhalten zur Laufzeit durch geeignete Maßnahmen sichergestellt werden. Hierbei kommt der Reaktionszeit für die Erkennung und Unterbindung von fehlerhaftem Verhalten besondere Bedeutung zu. Umso schneller ein fehlerhaftes Verhalten unterbunden werden kann, desto geringer ist die Beeinflussung von anderen Anwendungen.

Kann durch einen Separierungsmechanismus eine obere Schranke für die maximale Beeinflussung garantiert werden, kann diese obere Schranke für die Analyse verwendet werden und die Anforderungen an weniger kritische Anwendungen können auf einem niedrigen Level belassen werden. In diesem Fall muss nur der Separierungsmechanismus denselben hohen Anforderungen wie die Anwendung mit der höchsten Kritikalität genügen. Die im Rahmen der vorliegenden Arbeit entwickelten Hardware-Mechanismen, um die gegenseitige Beeinflussung auf ein festgelegtes Maß zu begrenzen, werden in Kapitel 4 beschrieben.

3.1. Gemeinsam verwendete Komponenten

Die zeitliche Planbarkeit (engl. *Schedulability*) eines Systems mit gemeinsam verwendeten Komponenten lässt sich verifizieren, indem man die spätesten Antwortzeiten (engl. *Worst-Case Response Time (WCRT)*) aller beteiligen Aufgaben zur Entwicklungszeit analysiert. Hierdurch lässt sich sicherstellen, dass alle kritischen Aufgaben vor einer gegebenen Frist (engl. *Deadline*) abschlossen werden können. Um eine solche Analyse durchführen zu können, muss das Verhalten aller Anwendungen im schlechtesten Fall bekannt sein. Dieses Wissen kann man beispielsweise dadurch erhalten, indem man sämtliche Anwendungen zunächst exklusiv auf einer Plattform ausführt und deren Verhalten beobachtet.

Die Beeinflussung einer Aufgabe durch eine gemeinsame Nutzung einer Komponente mit anderen Aufgaben wurde von Schliecker et al. in [78] analysiert. Deren Ansatz verwendet die *Busy-Window*-Methode, bei der die größtmöglichen Verzögerungen einer Aufgabe aufsummiert werden. Die maximale Anzahl an Zugriffen auf eine gemeinsam verwendete Komponente durch andere Aufgaben geht dabei multipliziert mit der Dauer der Zugriffe als zusätzliche Verzögerung in die Antwortzeit der analysierten Aufgabe ein.

Die maximale Verzögerung einer Aufgabe T_i durch andere Aufgaben, welche dieselbe gemeinsame Komponente verwenden, kann über

$$D_{i,S} = \sum_{p \in P} \tilde{\eta}_{p \to S}(\Delta t) \cdot m \tag{3.1}$$

beschrieben werden. Hierbei ist P die Menge aller Kacheln im System, von wo auf die gemeinsame Komponente S zugegriffen wird. $\tilde{\eta}_{p \to S}(\Delta t)$ ist die maximale Anzahl an Zugriffen auf die gemeinsam verwendete Komponente S durch Kachel p in einem beliebigen Zeitfenster der Größe Δt und m ist die Zeit, welche für einen Zugriff benötigt wird.

In der ursprünglichen Gleichung von Schliecker et al. bezeichnet P die Menge aller anderen Prozessoren eines Mehrkernprozessors, welche auf eine gemeinsam verwendete Komponente zugreifen.

Da auf der IDAMC-Plattform, welche in Kapitel 4 genauer beschrieben wird, eine Separierung von Anwendungen auf unterschiedlichen Kacheln, welche auch mehrere Prozessoren enthalten können, unterstützt wird, jedoch nicht die Separierung von Anwendungen auf unterschiedlichen Prozessoren derselben Kachel, werden in Gleichung 3.1 nicht Zugriffe von einzelnen Prozessoren, sondern von Kacheln mit möglicherweise mehreren Prozessoren betrachtet.

Wenn dann zur Laufzeit von einer Kachel p, welche weniger kritische Aufgaben ausführt, mehr Zugriffe als die bei der Analyse einer kritischen Aufgabe auf einer anderen Kachel angenommene Anzahl $\tilde{\eta}_{p\to S}(\Delta t)$ auf eine gemeinsam verwendete Komponente ausgeführt werden, zum Beispiel durch einen Fehler, kann die zusätzliche Verzögerung dazu führen, dass zugesicherte Garantien an die kritische Aufgabe nicht mehr eingehalten werden können und, als Folge davon, die kritische Aufgabe ihre Frist verpasst und fehlschlägt. Damit Garantien an kritische Aufgaben, welche aus der Analyse abgeleitet wurden, auch zur Laufzeit Gültigkeit behalten, muss die Verwendung von gemeinsamen Komponenten durch weniger kritische Aufgaben ebenfalls denselben hohen Anforderungen wie die analysierte kritische Aufgabe entsprechen. Eine Zertifizierung entsprechend einem hohen Sicherheitsintegritätslevels, obwohl ohne gemeinsam verwendete Komponenten lediglich ein niedriges Level erfüllt würden müsste, würde die Kosten einer gemeinsamen Plattform unverhältnismäßig stark ansteigen lassen.

Alternativ könnte man für alle Kacheln $p \in P$, die dieselbe Komponente S ebenfalls verwenden, die maximal mögliche Beeinflussung annehmen, also das theoretische Maximum an Zugriffen $\tilde{\eta}_{p\to S}(\Delta t) \to \infty$ in einem beliebigen Zeitfenster der Größe Δt. Dies bedeutet nicht notwendigerweise, dass die Plattform dann nicht mehr zeitlich planbar wäre. Durch genauere Betrachtung der verwendeten Plattform kann die maximal mögliche Anzahl an Zugriffen eventuell reduziert werden. Man würde beispielsweise annehmen, dass von allen anderen Kacheln ständig auf die gemeinsame Komponente zugegriffen wird. Diese Anzahl ist jedoch durch die Verzögerungen der Verbindungen zwischen den anderen Kacheln und der gemeinsam genutzten Komponente, durch die Zeit für benötigte Schaltvorgänge innerhalb der Prozessoren sowie durch die verwendeten Arbitrierungsmechanismen begrenzt. Dies würde die Planbarkeit eventuell verbessern, würde aber immer noch zu einem stark überdimensionierten System führen.

Die in der vorliegenden Arbeit in Abschnitt 4.2.10 beschriebenen Überwachungsmechanismen in den Netzwerkschnittstellen zwischen den Kacheln und dem NoC stellen zur Laufzeit sicher, dass die bei der Analyse angenommene Anzahl an Zugriffen, welche von den jeweiligen Kacheln auf eine gemeinsam genutzte Komponenten ausgeführt werden, nicht überschritten wird. In diesem Fall muss lediglich der Überwachungsmechanismus die Anforderungen eines hohen Sicherheitsintegritätslevel erfüllen, die weniger kritischen Aufgaben jedoch nicht.

Die Begrenzung der Zugriffe auf exakt die erwartete Anzahl $\tilde{\eta}_{p\to S}(\Delta t)$ ist durch die Reaktionszeit eines Überwachungsmechanismus auf einer realen Plattform jedoch oft nicht möglich. Das gilt vor allem dann, wenn, wie in anderen Lösungen [36], eine Interaktion mit einer zentralen Steuereinheit nötig ist, um weitere Zugriffe zu verhindern. Es mag jedoch oft auch ausreichen, wenn irgendeine obere Grenze für die Anzahl an Zugriffen $\tilde{\eta}^u_{p\to S}(\Delta t)$, welche von anderen Kacheln auf eine gemeinsam verwendete Komponente ausgeführt werden, garantiert werden kann. In dem Fall würde dann die daraus resultierende maximale zusätzliche Verzögerung einer kritischen Aufgabe T_i über

$$D^u_{i,S} = \sum_{p\in P} \tilde{\eta}^u_{p\to S}(\Delta t) \cdot m \tag{3.2}$$

bestimmt werden können.

Auf einem nicht voll ausgelasteten System mag es sogar wünschenswert sein, einen höheren Wert für $\tilde{\eta}^{u}_{p\to S}(\Delta t)$ anzunehmen, um so leichte Abweichungen von weniger kritischen Aufgaben vom spezifizierten Verhalten zuzulassen, solange während der Analyse alle kritischen Aufgaben ihre Fristen einhalten können.

Abbildung 3.1 zeigt ein Beispiel einer kritischen Aufgabe T1, welche einen Speicher mit einer weniger kritischen Aufgabe T2 gemeinsam verwendet. Der Speicher befindet sich in diesem Beispiel in einer dritten Kachel. Abbildung 3.1 oben zeigt zunächst die Antwortzeit $R^{max}_{T_1,\neg S}$ der Aufgabe T1, wenn diese den Speicher exklusiv verwendet. In Abbildung 3.1 Mitte ist die weniger kritische Aufgabe T2 ebenfalls aktiviert. Sie verhält sich zunächst so wie erwartet. Durch die Zugriffe einer weiteren Aufgabe auf denselben Speicher verlängert sich die Antwortzeit von T1 wie oben beschrieben auf $R^{max}_{T_1,\neg S} + D_{T_1,S}$. In Abbildung 3.1 unten greift Aufgabe T2, beispielsweise durch einen Fehler oder unerwartete Eingangsdaten, öfter auf den gemeinsamen Speicher zu als spezifiziert. Die Antwortzeit $R^{max}_{T_1,\neg S} + D^{u}_{T_1,S}$ verlängert sich erneut um die zusätzliche Anzahl an Zugriffen, bleibt aber gerade noch innerhalb der Frist. Der Fall einer weiteren Verzögerung wird Abschnitt 3.4 dargestellt.

In dem Beispiel wird nun angenommen, dass die späteste Antwortzeit der kritischen Aufgabe T1 bei exklusiver Nutzung des Speichers in Kachel 3 bei $0,8\,ms$ liegt. Die Verzögerung durch die gemeinsame Nutzung mit einer korrekt arbeitenden Aufgabe T2 beträgt $D_{T_1,S} = 0,2\,ms$ und die Antwortzeit in dem Fall damit $1\,ms$. Die einzuhaltende Frist von Aufgabe T1 liegt bei $1,2\,ms$. Die maximal erlaubte Verzögerung durch die gemeinsame Nutzung des Speichers beträgt also $D^{u}_{T_1,S} = 0,4\,ms$. Wenn nun ein Speicherzugriff $m = 400\,ns$ benötigt, berechnet sich die maximale Anzahl an zusätzlichen, über die erwartete Anzahl hinausgehenden erlaubten Zugriffen $\tilde{\eta}^{u}_{p\to S}(\Delta t) - \tilde{\eta}_{p\to S}(\Delta t)$ durch Aufgabe T2 in einem Zeitfenster von $\Delta t = 1,2\,ms$ unter Verwendung von Gleichung 3.1 und Gleichung 3.2 zu $(D^{u}_{T_1,S} - D_{T_1,S})/m = 500$.

$\tilde{\eta}^{u}_{p\to S}(1,2\,ms) = \tilde{\eta}_{p\to S}(1,2\,ms) + 500$ ist damit die maximale Anzahl an Zugriffen von der weniger kritischen Aufgabe T2 auf den gemeinsam verwendeten Speicher, welcher von einem geeigneten Separierungsmechanismus garantiert werden können muss, um von einer ausreichenden Separierung der beiden unterschiedlich kritischen Aufgaben sprechen zu können. Mehr zu der erlaubten Reaktionszeit eines Überwachungsmechanismus wird in Abschnitt 3.4 erläutert.

3.2. Kommunikation

Die Kommunikation zwischen Anwendungen auf unterschiedlichen Rechenkernen einer gemeinsamen Plattform sowie die Benachrichtigung durch ein verbundenes Peripheriemodul wird auf vielen Plattformen, wie auch auf der in der vorliegenden Arbeit vorgestellten IDAMC-Plattform, über Interrupt-Anfragen realisiert. Eingehende Interrupt-Anfragen aktivieren auf dem Zielprozessor eine Unterbrechungsroutine (engl. *Interrupt Service Routine (ISR)*), wodurch jedoch das Zeitverhalten aller anderen auf dem Zielprozessor ausgeführten Aufgaben beeinträchtigt werden kann. Interrupts können selbst hoch kritische Aufgabe unterbrechen, wenn Interrupts nicht bis zu dem entsprechenden Level deaktiviert sind.

3.2.1. Einfluss auf die Analyse

Um die Fristen aller auf dem Zielprozessor ausgeführten Aufgaben, besonders von kritischen Aufgaben, zur Laufzeit garantieren zu können, muss das System zunächst zur Entwicklungszeit analysiert

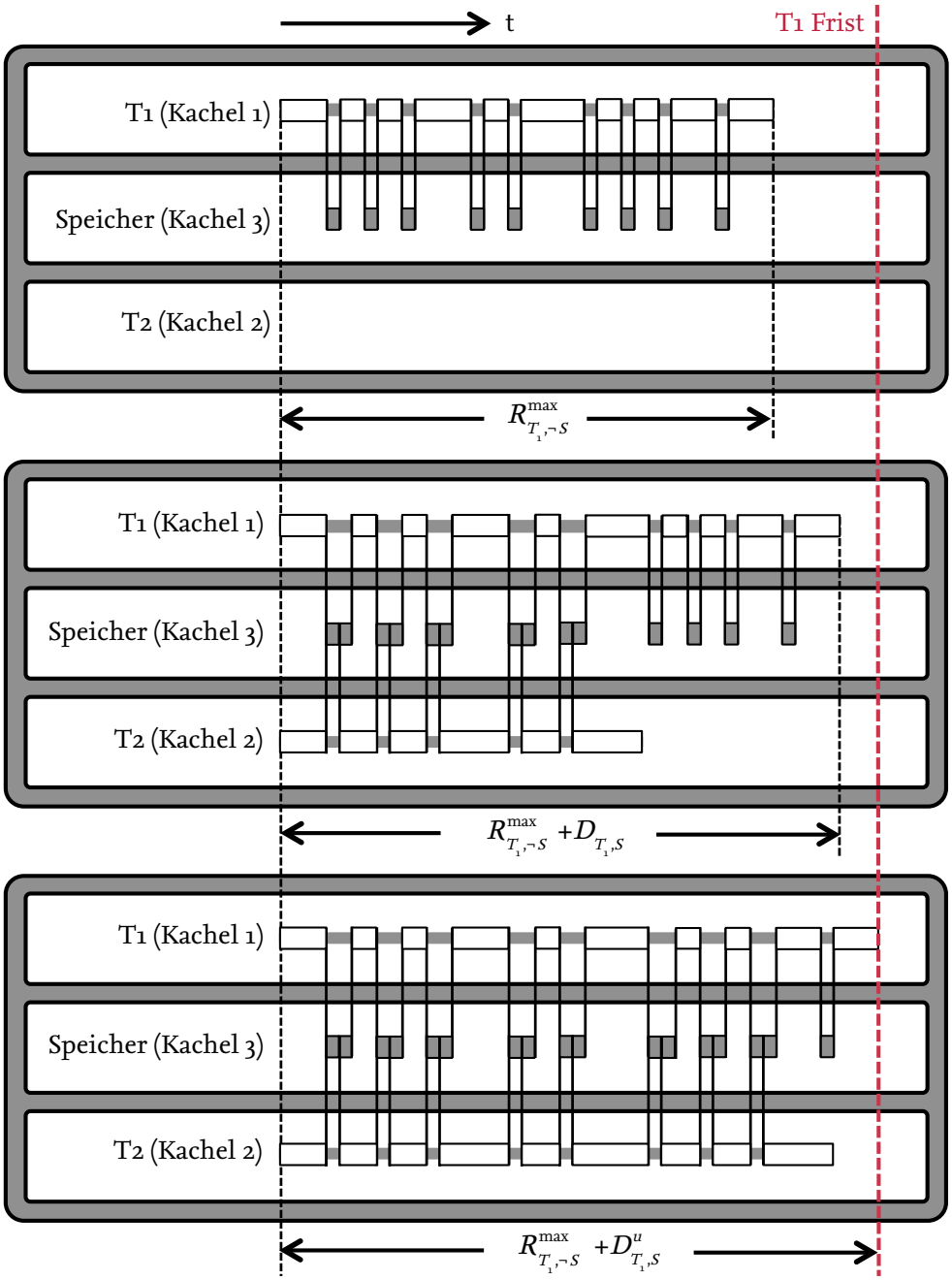

Abbildung 3.1.: Beispiel für einen gemeinsam verwendeten Speicher (in Anlehnung an [78])

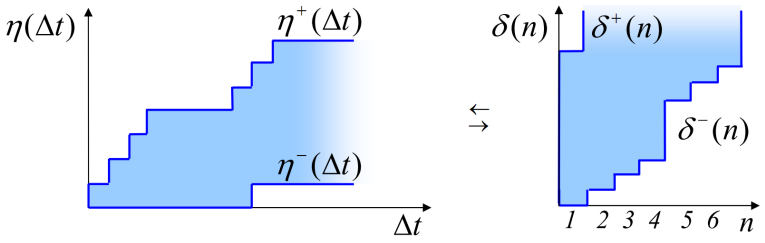

Abbildung 3.2.: Funktionen zur Modellierung von Ereignissen [77]

werden. Hierbei gehen Interrupt-Anfragen als weitere aufgabenaktivierende Ereignisse in die Analyse ein und können über Ereignismodelle beschrieben werden. Ereignismodelle können über die Ereignisfunktionen $\eta^+(\Delta t)$ und $\eta^-(\Delta t)$ beziehungsweise über die Abstandsfunktionen $\delta^+(n)$ und $\delta^-(n)$ definiert werden. Die Ereignisfunktionen $\eta^+(\Delta t)$ und $\eta^-(\Delta t)$ geben hierbei die maximale und minimale Anzahl von Ereignissen in einem beliebigen Zeitintervall Δt an und die Abstandsfunktionen $\delta^+(n)$ und $\delta^-(n)$ den maximalen und minimalen Abstand zwischen $n \geq 2$ aufeinanderfolgenden Ereignissen [45]. Die Funktionen können, wie in Abbildung 3.2 dargestellt, ineinander überführt werden. Für die sichere Separierung von unterschiedlich kritischen Aufgaben sind vor allem die maximale Anzahl beziehungsweise der minimale Abstand von Interesse, um die Separierung auch unter den schlechtesten Bedingungen garantieren zu können.

Zusammen mit den Ausführungszeiten aller Aufgaben sowie der angewendeten Methode für die Ablaufsteuerung (engl. *Scheduling Policy*) kann das System dann analysiert werden. Die Ergebnisse der Analyse sind jedoch nur dann gültig, wenn sich das System zur Laufzeit so verhält wie bei der Analyse angenommen, also nur dann, wenn Ereignisse entsprechend ihrem Modell auftreten. In [64] wurde ein Überwachungsmechanismus für Aktivierungsmuster zur Laufzeit als Teil des Betriebssystems auf einem Einzelprozessorsystem integriert.

Falls eingehende Nachrichten von anderen Rechenkernen oder Interrupt-Anfragen von Peripheriemodulen zur Laufzeit nicht dem angenommenen Ereignismodell entsprechen, zum Beispiel durch einen Fehler oder ein anderes nicht spezifiziertes Verhalten einer weniger kritischen Aufgabe, kann eine kritische Aufgabe ihre Frist eventuell überschreiten und fehlschlagen. Abbildung 3.3 zeigt ein Beispiel, in dem ein Prozessor eine kritische Aufgabe T1 ausführt und über Interrupts mit einer weniger kritischen Aufgabe auf einem anderen Prozessor kommuniziert. Beim Eintreffen eines neuen Interrupts wird eine Unterbrechungsroutine (ISR) ausgeführt, welche die kritische Aufgabe T1 jedes Mal unterbricht und dadurch deren Antwortzeit verlängert. Die Ausführungszeit der Unterbrechungsroutine beträgt jedes Mal zwei Zeiteinheiten. Die Ausführung der kritischen Aufgabe T1 beträgt 14 Zeiteinheiten und muss nach spätestens 25,5 Zeiteinheiten abschlossen sein. Der für die Analyse angenommene und in Abbildung 3.3 oben dargestellte Abstand zwischen zwei aufeinanderfolgenden Interrupts ist über die Abstandsfunktion mit $\delta^+(2) = 5$ definiert. Wenn nun zur Laufzeit, wie in Abbildung 3.3 unten dargestellt, dieser Abstand mit lediglich vier Zeiteinheiten zwischen aufeinanderfolgenden Interrupts unterschritten wird, können zur Entwicklungszeit gegebene Garantien nicht mehr eingehalten werden. In dem Beispiel wird die kritische Aufgabe T1 öfter als während der Analyse angenommen unterbrochen und damit so weit verzögert, dass sie nicht vor der Frist beendet werden kann.

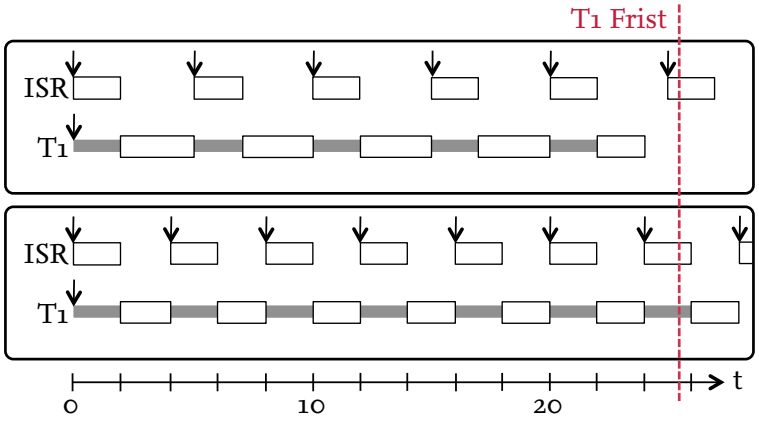

Abbildung 3.3.: Verzögerung durch Unterbrechungsroutine

Die Abweichung in dem Beispiel kann beispielsweise dadurch hervorgerufen werden, dass die weniger kritische Aufgabe nicht dieselben hohen Anforderungen wie die hoch kritische Aufgabe auf dem Zielprozessor erfüllen muss. Um also durch die Analyse einer hoch kritischen Aufgabe, deren späteste Antwortzeit garantieren zu können, müssen alle Eingangsparameter, wie Aktivierungs- oder Abstandsfunktionen aller anderen Aufgaben auf demselben Prozessor, Unterbrechungsroutinen eingeschlossen, denselben hohen Anforderungen genügen. Damit müssen sämtliche Aufgaben, welche über Interrupts mit einem Prozessor kommunizieren, der unter anderem kritische Aufgaben ausführt, ebenfalls die Anforderungen der höchsten Kritikalitätsstufe erfüllen, auch wenn diese Aufgaben an sich nicht kritisch sind. Dies würde, wie weiter oben bereits erläutert, die Entwicklungskosten für weniger kritische Aufgaben und Peripheriemodule und damit auch die Gesamtkosten einer Vielkernplattform mit unterschiedlich kritischen Aufgaben stark erhöhen.

Alternativ könnte man, wie bei der Verwendung gemeinsamer Komponenten auch, sämtliche Fehler und mögliche Abweichungen in der Analyse berücksichtigen, was aber auch hier zu einer weiteren Überdimensionierung des Systems führen würde. In der vorliegenden Arbeit wird der Ansatz verfolgt, das bei der Analyse angenommene Interrupt-Aufkommen zur Laufzeit zu erzwingen. Ein Mechanismus zur Überwachung des Interrupt-Aufkommens, welcher dezentral in den Netzwerkschnittstellen der Interrupt-Quellen implementiert ist, wird in Abschnitt 4.2.10 beschrieben. In diesem Fall muss lediglich der Separierungsmechanismus die Anforderungen der höchsten Kritikalitätsstufe erfüllen, die weniger kritischen Anwendungen, die mit höher kritischen Anwendungen kommunizieren, jedoch nicht.

3.2.2. Begrenzung des Interrupt-Aufkommens

Der Unterschied zwischen Interrupts und anderen Ereignissen im System ist die zeitliche Ablaufsteuerung, welche für Interrupts und normale Aufgaben eines Prozessors unabhängig voneinander durchgeführt wird. Eine Interrupt-Steuerungseinheit in Hardware maskiert und priorisiert zunächst alle aktiven Interrupts und leitet anschließend den Interrupt mit der höchsten Priorität an den zugehörigen Prozessor weiter. Dort wird dann die dem entsprechenden Interrupt zugeordnete Unterbrechungsroutine aufgerufen, welche selbst hoch kritische Aufgaben unterbrechen kann, wenn Interrupts nicht zuvor bis zu demselben hohen Level deaktiviert worden sind. Hoch kritische Aufga-

ben dagegen können Interrupts deaktivieren, jedoch keine Unterbrechungsroutinen unterbrechen. Diese gegenseitige Beeinflussung beeinträchtigt die Vorhersagbarkeit des Systems [37].

Leyva-del-Foyo et al. [37] haben hierfür eine softwarebasierte Lösung entwickelt, welche ohne eine zusätzliche Hardware-Unterstützung auskommt. Indem die Autoren den Großteil einer Unterbrechungsroutine in eine herkömmliche Aufgabe verlagern, welche dann bei einer eingehenden Interrupt-Anfrage durch die verkürzte Unterbrechungsroutine lediglich aktiviert wird, kann die für eine Unterbrechungsroutine benötigte Zeit minimiert werden. Die durch die Unterbrechungsroutine aktivierte Aufgabe wird dann anschließend wie jede andere Aufgabe des Systems behandelt. Auch wenn die Unterbrechungszeit pro eingehendem Interrupt in der Lösung minimiert werden kann, ist das System immer noch durch eine unbegrenzte Anzahl an möglichen Interrupts nicht vorhersagbar. Die Autoren der Arbeit schlagen die Verwendung eines sporadischen Servers vor, um Interrupts bei Überschreitung einer bestimmten Grenze deaktivieren zu können.

Ein sporadischer Server bekommt ein Budget zugewiesen, welches bei Ausführung des Servers verbraucht wird und sich nach einer gewissen Zeit wieder erneuert. Ist dieses Budget erschöpft, werden entsprechende Aufgaben nicht mehr ausgeführt bis das Budget wieder aufgefüllt ist. Dieser Ansatz wurde in [33] durch die Begrenzung der Rechenzeit, welche durch Interrupt-Anfragen auf dem zugehörigen Prozessor konsumiert werden darf, auf einen bestimmten Wert, die Interrupt-Server-Bandbreite, realisiert. Normale Aufgaben werden unabhängig von den Regeln des Interrupt-Servers ausgeführt. Hierdurch sollen herkömmliche Treiber mit einem unvorhersagbaren Zeitverhalten für Echtzeitsysteme einsetzbar gemacht werden.

Ein weiterer Ansatz, um einen Prozessor vor Überladung mit Interrupt-Anfragen zu schützen, wurde im PharOS-Betriebssystem implementiert [57]. Eingehende Interrupt-Anfragen werden zunächst gefiltert, indem die Anzahl an Interrupts in einem festgelegten Zeitintervall aufsummiert wird. Wird ein zuvor festgelegter Wert überschritten, werden entsprechende Interrupts deaktiviert.

Um zu verhindern, dass hoch kritische Aufgaben durch weniger kritische Interrupt-Anfragen unterbrochen werden, wurde in [76] dem eigentlichen Prozessor ein Co-Prozessor zur Seite gestellt. Dieser empfängt zunächst alle Interrupt-Anfragen für den Hauptprozessor und leitet nur solche weiter, welche eine höhere Priorität als die aktuell ausgeführte Aufgabe auf dem Hauptprozessor haben. Eine weitere Hardware-Lösung wurde von Strnadel et al. [82] auf einem separaten FPGA implementiert. Deren Interrupt-Steuerung kümmert sich um externen Interrupt-Anfragen an den Prozessor und begrenzt deren Anzahl soweit, dass eine sicherheitskritische Anwendung auf dem Prozessor nicht negativ beeinflusst werden kann. Hierfür kommuniziert die Interrupt-Steuerung auf dem FPGA mit dem Prozessor über eine proprietäre Verbindung, um ungenutzte Zeiten auf dem Prozessor für eine optimale Interrupt-Abarbeitung auszunutzen.

Einige Fehlerbedingungen und Situationen, in denen mehr Interrupt-Anfragen als erwartet auftreten, werden in [72] beschrieben. Die Autoren beschreiben, analysieren und vergleichen einige Software- und Hardware-Lösungen, um die Anzahl an Interrupt-Anfragen zu begrenzen. Alle Lösungen basieren jedoch auf dem Zählen von Interrupts in einem zugehörigen Zeitfenster an dem Ziel des Interrupts.

Für eine Vielkernplattform, bei der die verteilten Komponenten über ein gemeinsames NoC verbunden sind, ist die Begrenzung des Interrupt-Aufkommens an dem Interrupt-Ziel eher ungeeignet, da in diesem Fall Interrupt-Anfragen weiterhin an der Quelle erzeugt werden. Das erhöhte Aufkommen an Interrupt-Nachrichten wird daher immer noch durch das Netzwerk geschickt und kann dort

vermehrt zu Konkurrenzsituationen mit anderen möglicherweise auch kritischen Paketen führen. Werden diese übermäßig verschickten Interrupt-Pakete dann an dem Ziel nicht mehr akzeptiert ohne verworfen zu werden, kann dies zusätzlich zu einer Blockierung von Puffern in den Switches führen. Weiterhin ist es in diesem Fall nicht möglich, einzelne fehlerhafte Interrupt-Quellen zu deaktivieren, wenn die entsprechenden Interrupt-Nummern mit anderen Interrupt-Quellen geteilt werden.

Um eine kritische Kommunikation anderer Anwendungen im NoC im Fehlerfall durch ein erhöhtes Interrupt-Aufkommen nicht zu beeinträchtigen, könnte dieses erhöhte Aufkommen anstatt durch Filtern am Interrupt-Ziel besser direkt an der Quelle unterbunden werden. Hierbei sollte die Erkennung der Abweichung ebenfalls an den Quellen der Interrupts und nicht an den Zielprozessoren geschehen, um so die Reaktionszeit auf ein fehlerhaftes Verhalten zu verkürzen. Warum die Reaktionszeit für eine effizient Nutzung einer Vielkernplattform wichtig ist, wird in Abschnitt 3.4 erläutert.

Die Reduzierung der Reaktionszeit durch das Filtern von ausgehenden Interrupts an der Quelle durch einen Software-Mechanismus ist jedoch nur für die Kommunikation zwischen Prozessoren möglich. Das Interrupt-Aufkommen von verteilten Peripheriemodulen, ohne die Möglichkeit Software auszuführen, kann so nicht überwacht werden. In [10, 67] werden Hardware-Module beschrieben, welche es erlauben, kommerzielle Standardkomponenten (engl. *Commercial Of-The-Shelf (COTS)*) an ein Echtzeitsystem anzubinden. Kommerzielle Standardkomponenten erfüllen im Allgemeinen nicht die Anforderungen eines kritischen Echtzeitsystems. Durch Hardware-Brücken sollen diese Komponenten in den genannten Arbeiten jedoch echtzeitfähig gemacht werden. Ein weiteres zentrales Hardware-Module steuert bei dem vorgestellten Mechanismus die zeitliche Abfolge von Bus- und Speicherzugriffen sowie Interrupt-Anfragen durch die verteilten Echtzeitbrücken wie Aufgaben auf einem Prozessor. Durch die benötigte zentrale Instanz und die damit verbundene mangelnde Skalierbarkeit ist der beschriebene Mechanismus jedoch für Vielkernplattformen mit möglicherweise sehr vielen Peripheriemodulen eher ungeeignet. Weiterhin würden sich Zugriffe auf eine zentrale Steuerung, um Peripheriemodule zu einzelnen Anwendungen zuzuweisen und deren zeitliche Ablaufsteuerung zu konfigurieren, auf Plattformen mit unterschiedlich kritischen Anwendungen nicht ausreichend separieren lassen.

3.3. Energieverbrauch

Es bestehen zwei Risiken in Bezug auf den Energieverbrauch unterschiedlich kritischer Anwendungen auf einer Vielkernplattform. Zum einen kann eine weniger kritische Anwendung durch einen zu hohen Energieverbrauch die meist begrenzt vorhandene Energie für kritische Anwendungen so weit reduzieren, dass eine sichere Ausführung nicht mehr garantiert werden kann. Zum anderen kann eine hohe lokale Energiedichte durch eine erhöhte Aktivität in einem begrenzten Bereich des Chips zu einer lokalen Überhitzung führen (engl. *Hot Spot*). Die lokale Überhitzung kann die Ausführung in benachbarten Bereichen beeinflussen und sogar zu einer Verringerung der Verfügbarkeit und Lebensdauer des gesamten Chips führen [79]. Es ist also wichtig, den Gesamtenergieverbrauch einzelner Anwendungen sowie die von diesen Anwendungen ausgehende lokale Energiedichte zur Entwicklungszeit zunächst zu analysieren und zur Laufzeit sicherzustellen, dass das tatsächliche Verhalten mit dem analysierten Verhalten übereinstimmt.

Die Gesamtverlustleistung eines Chips lässt sich über einen dynamischen und einen statischen Anteil angeben. Der statische Anteil wird hauptsächlich durch Leckströme hervorgerufen und der dynamische Anteil kann weiter aufgeteilt werde, in Verlustleistung durch Schaltvorgänge und Verlustleistung durch Kurzschlussströme. Der Zusammenhang lässt sich über die Formel

$$P = P_{dyn} + P_{sta} = P_{swi} + P_{sc} + P_{leak} \tag{3.3}$$

wiedergeben. Der größte Anteil an der Verlustleistung in CMOS-Schaltungen wird hierbei durch Schaltströme hervorgerufen. Die Schaltverlustleistung ist durch die Formel

$$P_{swi} = \alpha \cdot C_L \cdot V^2 \cdot f \tag{3.4}$$

definiert, wobei V die Versorgungsspannung und C_L parasitäre Kapazitäten sind, welche mit der Frequenz f umgeladen werden, sofern die Schaltung aktiv ist [19]. Die Schaltrate α und damit auch der dynamische Anteil an der Verlustleistung ist hierbei abhängig von der ausgeführten Anwendung und dessen Eingangsdaten [66]. Ein fehlerhaftes oder nicht spezifiziertes Verhalten von einer Anwendung kann damit zu einer nicht vorhersagbaren Verlustleistung beziehungsweise Energiedichte führen.

Um einen erhöhten Energieverbrauch einer weniger kritischen Anwendung oder eine von dieser Anwendung verursachte erhöhten Energiedichte erkennen zu können, um eine ausreichende Separierung zu kritischen Anwendungen auf derselben Plattform sicherzustellen, muss zum einen der Energieverbrauch für jede Anwendung und zum anderen die Energiedichte für einzelne Chip-Regionen individuell bestimmbar sein. Da die interne Logik eines Chips meist durch eine einzelne oder nur sehr wenige Spannungsversorgungen gespeist wird, bietet sich hierbei durch physikalische Messungen lediglich die Möglichkeit, die Leistung der gesamten internen Logik zu bestimmen, jedoch nicht von individuellen Anwendungen oder in einzelnen Regionen. Selbst Vielkernprozessoren mit mehreren Spannungsinseln, wie beispielsweise Intels SCC, erlauben nur die Messung des Gesamtstromverbrauchs und damit der Gesamtverlustleistung aller Kacheln des Systems [50]. Der Energieverbrauch pro Kachel lässt sich beim SCC nicht bestimmen. Selbst wenn dies möglich wäre, lässt sich der Energieverbrauch in gemeinsam verwendeten Komponenten nicht den jeweiligen Anwendungen zuordnen [15], was wichtig wäre, um einen erhöhten Energieverbrauch beziehungsweise eine erhöhte Energiedichte durch weniger kritische Anwendungen zu vermeiden, ohne dabei kritische Anwendungen zu beeinflussen.

Ein kommerzieller Mehrkernprozessor mit Mechanismen, um einer erhöhten Hitzeentwicklung entgegenzuwirken, ist IBMs Cell-Prozessor [68]. Eine Temperatursteuereinheit (engl. *Thermal Management Unit (TMU)*) auf dem Chip überwacht die Gesamttemperatur des Chips und zusätzlich die individuellen lokalen Temperaturen in kritischen Regionen. Die Messung der Gesamttemperatur wird zur Steuerung einer externen Kühlung verwendet, wie beispielsweise einem Lüfter, wie sie auf vielen Standardprozessoren eingesetzt werden. Die Messungen der lokalen Temperaturen werden verwendet, um einzelne Rechenkerne zu drosseln oder abzuschalten, bis hin zur Abschaltung des gesamten Chips, um permanente Schäden zu verhindern.

Xilinx' Virtex-6-FPGA-Familie [94] ist ein weiteres kommerzielles Standardprodukt mit der Möglichkeit, Temperatur und Leistung zur Laufzeit zu überwachen. Ein Systemmonitor [95] mit einem eingebauten Analog-Digital-Umsetzer erlaubt die Überwachung der Chip-Temperatur, der Versorgungsspannung sowie weiterer externer analoger Eingänge. Einer dieser analogen Eingängen kann verwendet werden, um den Abfall der Versorgungsspannung über einem externen Shunt-Widerstand

zu bestimmen, um so die aktuelle Leistung überwachen zu können. Da die gesamte interne Logik eines Virtex-6-FPGAs, wie bei vielen anderen Chips auch, über einen einzelnen Spannungseingang versorgt wird, lässt sich auch hier lediglich die Leistung der gesamten internen Logik bestimmen und nicht die Verlustleistung für individuelle Komponenten des Chips. Darüber hinaus lässt sich bei Verwendung eines Virtex-6-FPGAs, wie beim Cell-Prozessor auch, die verursachende Anwendung von einem erhöhten Energieverbrauch beziehungsweise einer erhöhten lokalen Temperatur in einer geteilten Komponente nicht bestimmen [15].

Aus diesen Gründen sind physikalische Messungen ungeeignet, um zur Laufzeit den Energieverbrauch unterschiedlich kritischer Anwendungen auf einer geteilten Plattform zu separieren.

3.3.1. Ereignisbasierte Leistungsabschätzung

Um dennoch den individuellen Energieverbrauch von einzelnen Anwendungen sowie die Energiedichte in einzelnen Chip-Regionen zur Laufzeit ermitteln zu können, kann man sich zunutze machen, dass in Gleichung 3.4 sowohl die parasitären Kapazitäten, die Versorgungsspannung als auch die Frequenz bekannt sind und die Schaltverlustleistung damit dann nur noch von der Aktivität α der Schaltung abhängt. Bellosa et al. [16] entdeckten eine starke Korrelation zwischen Ereignissen im System, wie Fließkommaoperationen oder Cache-Fehltreffer, welche auf Aktivität in bestimmten Einheiten des Chips hindeuten, und der aktuell benötigten Energie. Durch Aufsummieren von bestimmten Ereignissen mittels für die Überwachung der Rechenleistung vorgesehenen Zählern (engl. *Performance Monitoring Counter (PMC)*) eines Intel-Pentium-II-Prozessors konnten die Autoren die Verlustleistung abschätzen. Solche PMCs finden sich in den verschiedensten Prozessoren zur Bestimmung der Rechenleistung, konnten aber auch erfolgreich mit einer Abweichung von bis zu unter 5% zur Abschätzung der Verlustleistung eingesetzt werden.

Die dynamische Verlustleistung eines Systems mit n PMCs kann dann durch Multiplikation der Zählerstände PMC_i mit der zugehörigen Energie e_i, dividiert durch das entsprechende Messintervall t_{sample}, abgeschätzt werden:

$$P_{dyn} = \frac{\sum_{i=1}^{n}(PMC_i \cdot e_i)}{t_{sample}} \quad . \tag{3.5}$$

Um die Energie für bestimmte Ereignisse des Systems zu ermitteln, können Mikrobenchmarks verwendet werden. Die Autoren von [16] verwendeten Mikrobenchmarks, welche lediglich Ereignisse derselben Art, wie beispielsweise Fließkommaoperationen, unterbrochen durch variable Ruhezeiten, enthielten. Die Autoren variierten die Ruhezeiten zwischen den Ereignissen und ermittelten zur Laufzeit die aktuelle Leistungsaufnahme des Systems. Sie entdeckten eine Korrelation zwischen der Länge der Ruhezeiten in den Mikrobenchmarks und der benötigten Gesamtenergie des Systems. Hieraus konnten die Autoren mittels linearer Regression die benötigte Energie pro Ereignis bestimmen.

Bhattacharjee et al. [18] wählten einen ähnlichen Ansatz. Die Autoren ersetzten jedoch reale Messungen am System durch Simulationen auf Gatterebene für die Charakterisierung der Ereignisse. Die durch Mikrobenchmarks hervorgerufene Aktivität der einzelnen Gatter der verwendeten Plattform wurde während der Simulation aufgezeichnet und unter Verwendung von Synopsys' Prime-Time PX dafür eingesetzt, um den hervorgerufenen Leistungsverbrauch in einzelnen Komponenten separat bestimmen zu können.

Die Charakterisierung energieintensiver Ereignisse ist notwendig, um zunächst den Energieverbrauch einzelner Komponenten modellieren zu können. Anschließend kann die Verlustleistung eines Systems, einzelner Komponenten oder auch individueller Anwendungen über das Zählen von Ereignissen abgeschätzt werden. Die Modelle beschreiben einzelne Komponenten oder Gruppen von Komponenten über ein oder mehrere aktivierende Ereignisse und der dafür benötigten Energie. Die Anzahl und Art der Ereignisse bestimmt hier zum einen die Genauigkeit der Abschätzung und zum anderen den nötigen Mehraufwand für die Leistungsabschätzung. Existierende Lösungen reichen von wenigen vorhandenen Zählern in Standardprozessoren zur Messung der Rechenleistung [30] bin hin zu Zählern für nahezu jede Gatteraktivität [29].

Bei den Standardprozessoren gibt es sowohl Einzelprozessorsysteme wie Intels Pentium-4 als auch Mehrkernprozessoren wie AMDs Phenom-II oder Intels Core-2-Duo, für welche die ereignisbasierte Verlustleistungsabschätzung erfolgreich umgesetzt werden konnte [53, 20, 17]. Durch die begrenzte Anzahl an PMCs von Standardprozessoren sind diese jedoch nur bedingt für die Verlustleistungsabschätzung zur Laufzeit geeignet. In [30] verwenden die Autoren die zwei verfügbaren PMCs eines Intel-PXA255-Prozessors, um fünf Ereignisse zu überwachen. In dem Fall sind mehrere Durchläufe eines Programms nötig, um mit lediglich zwei Zählern aber fünf Ereignissen eine komplette Abschätzung der Verlustleistung durchzuführen, was das Verfahren für eine Überwachung des Energieverbrauchs zur Laufzeit ausschließen würde.

Manche Autoren verwenden vorhandene Zähler zur Abschätzung der Verlustleistung [30], andere wiederum fügen der verwendeten Plattform dedizierte Zähler für das Zählen von energieintensiven Ereignissen hinzu [66, 18]. In [40] wird dem Prozessor des Systems ein Co-Prozessor speziell für die Abschätzung der Verlustleistung zur Seite gestellt. Dieser besteht hauptsächlich aus Zählern für energieintensive Ereignisse. Selbst aktuelle kommerzielle Systeme, wie Intels Sandy-Bridge-Mikroprozessor [73], bieten digitale Leistungsmessung basierend auf dedizierten Zählern zur Verfolgung von Aktivitäten in den Hauptkomponenten. Aber auch hier lässt sich die Verlustleistung in gemeinsam verwendeten Komponenten, wie beispielsweise Speicher oder Grafikprozessor, nicht den verursachenden Anwendungen zuordnen.

Da PMCs nicht für die Abschätzung der Verlustleistung vorgesehen sind, muss zusätzlicher Aufwand betrieben werden, um von den zählbaren Ereignissen auf die aktuelle Verlustleistung schließen zu können. Dies ist besonders kompliziert, wenn die Verlustleistung in Komponenten außerhalb der Prozessoren abgeschätzt werden soll. In [30] wurde beispielsweise die Anzahl an Cache-Fehltreffern dafür verwendet, um die durch Aktivitäten im Speicher hervorgerufene Verlustleistung abzuschätzen. Die Autoren von [25] fügten dagegen dem Bus eines Einzelprozessorsystems eine weitere Einheit hinzu, um den Energieverbrauch in Speichern über die Überwachung von Transaktionen auf dem gemeinsamen Bus abzuschätzen. Dieser Ansatz könnte auch dafür eingesetzt werden, um die Verlustleistung eines System-on-Chips (SoCs) aus existierenden Komponenten, ohne die Möglichkeit Ereignisse zu zählen, annähern zu können.

Für sicherheitskritische und unterschiedlich kritische Anwendungen kann es sogar sinnvoll sein, einer Komponente keine Zähler hinzuzufügen und selbst vorhandene Zähler nicht zu nutzen, da beispielsweise die Sicherheitsnorm IEC 61508 die Trennung von dem Überwachungsmechanismus und der zu überwachenden Funktion nahelegt. In diesem Fall kann der Energieverbrauch in einer Komponente über ihr externes Verhalten modelliert werden. Die Verlustleistung lässt sich dann über die Überwachung von relevanten Transaktionen auf dem Bus beziehungsweise dem NoC abschätzen.

Dieser Ansatz wird für die Laufzeitüberwachung des Energieverbrauchs von unterschiedlich kritischen Anwendungen auf der IDAMC-Vielkernplattform angewandt und in Abschnitt 4.2.10 näher erläutert.

Ein etwas anderer Ansatz zur Bestimmung der Verlustleistung zur Laufzeit wird in [58] beschrieben. Die Autoren fokussierten sich auf durch das Betriebssystem verursachte Verlustleistung und fanden eine starke Korrelation zwischen dem Energieverbrauch von Betriebssystemroutinen und der zugehörigen Anzahl an Befehlen pro Taktzyklus (engl. *Instructions Per Cycle (IPC)*). Ein hoher IPC spiegelt dabei einen hohen Anteil an aktiven Komponenten eines Prozessors wieder und damit einen höheren Energieverbrauch. Für die vorgestellte Abschätzung der Verlustleistung auf der Ebene von Betriebssystemroutinen musste deren Energieverbrauch zunächst charakterisiert werden. Hierfür wurden wie in [15] Mikrobenchmarks und lineare Regression verwendet. Die berechneten Parameter des Regressionsmodells $k0$ und $k1$ konnten anschließend an den Übergängen zwischen Betriebssystemroutinen aus einer Tabelle ausgelesen werden, um die Verlustleistung

$$P = k1 \cdot IPC + k0 \tag{3.6}$$

der entsprechenden Routine zur Laufzeit bestimmen zu können.

Die Abschätzung der Verlustleistung zur Laufzeit durch das Zählen von energieintensiven Ereignissen wurde in zahlreichen Anwendungen eingesetzt. Um die Simulationszeiten für die Analyse des Energieverbrauchs von komplexen SoCs zu reduzieren, wurde in [18] ein ASIC-Design mit Zählern versehen auf einem FPGA implementiert. Mit den tatsächlichen Energiegewichten für die ASIC-Bibliothek und den Zählerständen des FPGAs konnten die Autoren die Verlustleistung von beliebigen Anwendungen deutlich schneller abschätzen als mittels einer Simulation auf Gatterebene. Ein weiterer Ansatz, um die benötigte Energie für die Ausführung bestimmter Programme durch Emulation evaluieren zu können, bevor der Chip tatsächlich vorhanden ist, wird in [39] dargestellt. In [9] wurde darüber hinaus die automatische Charakterisierung von Ereignissen sowie die Generierung von Modellen für die Verlustleistungsabschätzung untersucht.

Weiterhin existieren einige Arbeiten in denen die Abschätzung der Leistung zur Laufzeit für die Verwaltung beziehungsweise die Optimierung der vorhandenen Energie eingesetzt wird. In [80] wird auf der Basis von gezählten Ereignissen eine dynamische Anpassung der Versorgungsspannung (engl. *Voltage Scaling*) angewendet und in [16] konnte eine energieoptimierte zeitliche Ablaufplanung der Aufgaben eines Systems entwickelt werden. Die Autoren von [17] verwenden PMCs zur Abschätzung der Verlustleistung, um eine Abrechnung nach individuellem Energieverbrauch für virtuelle Maschinen in Server-Farmen zu ermöglichen.

Mit einem erhöhten Energieverbrauch verbundene Probleme der vermehrten Hitzeentwicklung werden in [8] untersucht. Mit weiteren Gefährdungen durch eine hohe Verlustleistung einzelner Anwendungen und der sich daraus eventuell ergebenden lokalen Überhitzungen und möglichen Gegenmaßnahmen beschäftigten sich die Autoren von [18, 62]. In den Arbeiten werden Aufgaben von überhitzten Prozessoren auf kühlere Prozessoren migriert oder die Rechenleistung durch die Verringerung der Gesamtfrequenz reduziert. Chung et al. [26] erläutern wie sich mittels Aktivitätszählern eine Temperatur bestimmen lässt. Die Autoren konnten auf der Basis von PMCs feingranulare Überhitzungen feststellen, welche über reguläre Temperatursensoren nicht erkannt werden können.

In der vorliegenden Arbeit wird die ereignisbasierte Leistungsabschätzung verwendet, um die Gesamtverlustleistung einzelner Anwendungen sowie die Energiedichte innerhalb einzelnen Kacheln

zu bestimmen, um so den Energieverbrauch unterschiedlich kritischer Anwendungen auf einer gemeinsamen Plattform separieren zu können. Weiterhin lässt sich hierüber der individuelle Beitrag von Anwendungen zu der Energiedichte in gemeinsam verwendeten Kacheln ermitteln.

3.3.2. Analyse des Energiebedarfs

Die im Folgenden vorgestellte Analyse der dynamischen Verlustleistung, welche bereits in [2] veröffentlicht wurde, basiert auf der Analyse des zeitlichen Verhaltens, welche in [45] beschrieben wird. Die Autoren Henia et al. analysierten das zeitliche Verhalten eines Systems basierend auf Ereignisfunktionen $\eta_T^+(\Delta t)$ und $\eta_T^-(\Delta t)$, welche die Aktivierung von Aufgaben beschreiben, sowie für die Ausführung der aktivierten Aufgabe jeweils benötigten Zeit. $\eta_T^+(\Delta t)$ gibt hierbei die maximale Anzahl an aktivierenden Ereignissen für eine Aufgabe T an und $\eta_T^-(\Delta t)$ die minimale Anzahl an Ereignissen, welche in einem beliebigen Zeitintervall der Länge Δt auftreten können. Auf dem Modell aufbauend wurde in [69] eine Methode entwickelt, die Verlustleistung eines Systems über Frequenz- und Spannungsskalierung sowie Takt- und Spannungsabschaltung unter Berücksichtigung von Echtzeitbedingungen zu optimieren. Die vorgestellte Methode ist ein reines Verfahren zur Entwicklungszeit ohne dynamische Anpassungen zur Laufzeit. Die Analyse der Verlustleistung selbst wird in der Arbeit nicht beschrieben.

Schliecker et al. [78] erweiterten das Modell von Henia et al., wie in Abschnitt 3.1 erläutert, mit Zugriffen auf gemeinsam verwendete Komponenten $\tilde{\eta}_{T\to S}^+(\Delta t)$ und der angesammelten Belegungszeit (engl. *Aggregate Busy Time*). $\tilde{\eta}_{T\to S}^+(\Delta t)$ ist hierbei die maximale Anzahl an Anfragen an eine gemeinsam verwendete Komponente S durch eine Aufgabe T in einem Zeitfenster der Größe Δt und die angesammelte Belegungszeit summiert die Anzahl an Zugriffen und die dafür benötigte Zeit auf.

Für die hier beschriebene Analyse der Verlustleistung wird die benötigte Zeit pro Ereignis in der Analyse des Zeitverhaltens durch die verbrauchte Menge an Energie pro Ereignis ersetzt. Die benötigten Energiegewichtungen für die einzelnen Ereignisse können, wie bereits in [16, 18, 39] gezeigt wurde, über Mikrobenchmarks und lineare Regression bestimmt werden. Die notwendige Charakterisierung für die Ereignisse der in Kapitel 4 beschriebene IDAMC-Plattform wird in Abschnitt 5.2.1 erläutert.

Da die in den Komponenten der Plattform verbrauchte Energie über die Überwachung des externen Verhaltens einer Komponente oder von Transaktionen auf lokalen Bussen und des NoCs abgeschätzt wird, müssen zugehörige Energiegewichtungen für Komponenten, bei denen der Energieverbrauch vom internen Zustand oder von vorhergehenden Ereignissen abhängt, über ein Energieintervall modelliert werden. Angelehnt an die Notation von Schliecker et al. beschreiben $\tilde{e}_{T\to S}^+$ und $\tilde{e}_{T\to S}^-$ die maximale Energie beziehungsweise die minimale Energie, welche benötigt wird, wenn Aufgabe T auf die Komponente S zugreift. Beispielsweise wird ein Ereignis für eine Komponente im Ruhezustand mehr Energie benötigen als ein Ereignis für eine bereits aktive Komponente. Für eine Komponente im Ruhezustand muss zunächst Energie aufgewendet werden, um diese Komponente in den aktiven Zustand zu versetzen. Ein weiteres Beispiel ist das mehrmalige Schreiben auf dieselbe Speicherstelle. Für den Fall, dass zweimal dieselben Daten geschrieben werden, wird weniger Energie verbraucht als wenn unterschiedliche Daten geschrieben werde, da im ersten Fall keine Kapazitäten im Speicher umgeladen werden müssen.

Um nun Garantien bezüglich des Energieverbrauchs an einzelne Anwendungen geben zu können, ist vor allem eine sicher obere Grenze für den Energieverbrauch individueller Aufgaben sowie für

die Energiedichte in einzelnen Chip-Bereichen wichtig. Daher werden im Folgenden die oberen Grenzen $\tilde{e}^{+}_{T\to S}$ des Energieverbrauchs einzelner Ereignisse für die Analyse der Verlustleistung und die Zuordnung von Energiebudgets zu einzelnen Aufgaben verwendet.

Mit den Ereignisfunktionen $\tilde{\eta}^{+}_{T\to S}(\Delta t)$ und Energiegewichtungen $\tilde{e}^{+}_{T\to S}$ kann der dynamische Energieverbrauch der Gesamtmenge aller laufenden Aufgaben $\mathbf{T}$, welche Komponenten aus der Gesamtmenge an verfügbaren Komponenten $\mathbf{S}$ in einem beliebigen Zeitintervall der Länge Δt verwenden, über die Gleichung

$$E^{+}_{\mathbf{T}\to\mathbf{S}}(\Delta t) = \sum_{T\in\mathbf{T}} \left(\sum_{S\in\mathbf{S}} \tilde{\eta}^{+}_{T\to S}(\Delta t) \cdot \tilde{e}^{+}_{T\to S} \right) \tag{3.7}$$

ausgedrückt werden.

Die durchschnittliche dynamische Verlustleistung des analysierten Systems kann damit für den schlechtesten Fall mit

$$P^{+}_{dyn} = \frac{E^{+}_{\mathbf{T}\to\mathbf{S}}(\Delta t)}{\Delta t} \tag{3.8}$$

angegeben werden.

Um nun eine erhöhte lokale Energiedichte, welche zu einer Gefährdung durch Überhitzung führen könnte, in einer Region auf dem Chip erkennen zu können, muss die maximal mögliche Energiedichte in dieser Region zunächst bestimmt werden. Gleichung 3.7 kann hierfür verwendet werden, indem anstatt der Gesamtmenge aller Komponenten des Chips, lediglich die Untermenge $\mathbf{S}^{*} \subseteq \mathbf{S}$ aller Komponenten in dem analysierten Bereich, zum Beispiel in einer bestimmten Kachel, eingesetzt wird. Darüber hinaus können die beiden obigen Gleichung dafür eingesetzt werden, die durchschnittliche dynamische Verlustleistung für jede Anwendung individuell zu ermitteln. Hierfür müssen in die Gleichungen anstatt aller Aufgaben nur die Untermenge $\mathbf{T}^{*} \subseteq \mathbf{T}$ der Aufgaben der untersuchten Anwendung eingesetzt werden.

Mit dem Gesamtenergiebudget des Systems und den analysierten Maximalanforderungen der einzelnen Anwendungen kann bestimmt werden, ob das System sicher ausgeführt werden kann. Wenn sich jedoch zur Laufzeit das Verhalten, beispielsweise einer weniger kritischen Anwendung durch weniger strenge Auflagen, vom analysierten Verhalten unterscheidet, verliert die Analyse ihre Gültigkeit und Garantien können möglicherweise nicht mehr eingehalten werden. Damit die Anzahl der energieintensiven Ereignisse zur Laufzeit dem analysierten Verhalten entspricht, wie bereits in den Abschnitten 3.1 und 3.3 erläutert, könnten hierfür höhere Anforderungen an weniger kritische Aufgaben gestellt werden oder mögliche Fehler und Abweichungen in der Analyse betrachtet werden. Beide Möglichkeiten würden jedoch die Kosten einer gemeinsam von unterschiedlich kritischen Anwendungen verwendeten Plattform stark erhöhen.

In Abschnitt 4.2.10 wird ein Hardware-Mechanismus vorgestellt, der es erlaubt, das Gesamtenergiebudget auf einzelne Anwendungen und Regionen des Chips aufzuteilen und diese Teilbudgets zur Laufzeit zu überwachen und gegebenenfalls auch zu erzwingen. In dem Fall müsste, wie weiter oben bereits erläutert, der Separierungsmechanismus die Anforderungen des höchsten Kritikalitätslevels erfüllen, die weniger kritischen Anwendungen jedoch nicht.

3.4. Reaktionszeit

Um eine ausreichende Separierung zwischen unterschiedlich kritischen Anwendungen garantieren zu können, indem das in diesem Kapitel beschriebene analysierte Verhalten durch einen Überwa-

chungsmechanismus zur Laufzeit sichergestellt wird, ist es nicht nur wichtig ob, sondern auch wann ein Ausfall oder eine Abweichung einer weniger kritischen Aufgabe, welche zu einem Ausfall einer kritischen Aufgabe führen können, erkannt werden.

Nach der Sicherheitsnorm IEC 61508 [51] sind zwei Anwendungen voneinander abhängig, wenn, wie bereits in Abschnitt 2.1 erläutert, die Wahrscheinlichkeit für den Ausfall der einen Anwendung Einfluss auf die Wahrscheinlichkeit für den Ausfall der anderen Anwendung hat. Im Fall einer auf Kacheln basierenden Vielkernplattform, wie der in Kapitel 4 beschriebenen IDAMC-Plattform, bedeutet dies, falls sich ein Ausfall einer Anwendung auf einer der Kacheln als Fehler im NoC oder in anderen Kacheln ausbreitet und dort, falls er nicht rechtzeitig erkannt wird, zu einem Ausfall einer weiteren Anwendung führt, sind diese Anwendungen nicht unabhängig und damit nicht ausreichend separiert. Dies kann beispielsweise dadurch passieren, dass, wie weiter oben bereits erläutert, eine weniger kritische Aufgabe öfter als spezifiziert auf eine gemeinsam verwendete Komponente zugreift, mehr Nachrichten als angenommen an einen Prozessor mit einer kritischen Anwendung schickt oder durch eine erhöhte Aktivität zu einem erhöhten Energieverbrauch oder einer erhöhten Energiedichte führt. Wenn nun eine kritische Aufgabe dadurch beispielsweise eine Frist verpasst oder nicht zu Ende ausgeführt werden kann, sind diese Aufgaben abhängig.

In der IEC 61508 ist die Prozesssicherheitszeit, wie ebenfalls bereits in Abschnitt 2.1 beschrieben, definiert als der Zeitraum zwischen einem Ausfall, der zu einem gefährlichen Ereignis führen kann, und dem Zeitpunkt, zu dem dieser Ausfall erkannt und eine entsprechende Reaktion abgeschlossen sein muss, bevor dieses gefährliche Ereignis eintritt. Hiernach muss eine erhöhte Anzahl an Zugriffen auf eine gemeinsam verwendete Komponente, ein verringerter Abstand zwischen aufeinanderfolgenden Nachrichten oder eine erhöhte Aktivität erkannt und eine geeignete Reaktion abgeschlossen sein, bevor eine kritische Aufgabe auf einer anderen Kachel so weit verzögert wird, dass sie ihre Frist verpasst, beziehungsweise bevor die verbleibende Energie nicht mehr für die sichere Ausführung der kritischen Aufgabe ausreicht.

Abbildung 3.4 zeigt ein Beispiel, welches das Beispiel von Abbildung 3.1 fortführt. Sie zeigt die Reaktionszeit, um ein erhöhtes Aufkommen an Zugriffen durch die weniger kritische Aufgabe T2 auf einen gemeinsam verwendeten Speicher zu erkennen und zu unterbinden, bevor die kritische Aufgabe T1 ihre Frist verpasst. Auch wenn sich das Beispiel hier auf gemeinsam verwendete Komponenten bezieht, ist es genauso auf die Anzahl von eingehenden Nachrichten oder energieintensiver Ereignissen anwendbar.

Nachdem Abbildung 3.1 bereits die Verzögerung der Antwortzeit der kritischen Aufgabe T1 durch die Zugriffe der weniger kritischen Aufgabe T2 verdeutlicht hat, zeigt Abbildung 3.4 nun, wie ein fehlerhaftes Verhalten von T2 dazu führen kann, dass die kritische Aufgabe T1 durch eine weitere Verzögerung ihre Frist verpasst und wie lang die Reaktionszeit sein darf, um dies zu verhindern. In Abbildung 3.1 unten wurde bereits die maximale Anzahl an zusätzlichen Zugriffen durch die weniger kritische Aufgabe T2 gezeigt, damit die kritische Aufgabe T1 gerade noch vor ihrer Frist beendet werden kann. Diese maximal erlaubten zusätzlichen, über die erwartete Anzahl hinausgehenden Zugriffe werden in Abbildung 3.4 in orange dargestellt. Eine weitere Verzögerung durch den in rot dargestellten Zugriff würde bedeuten, dass eine Reaktion auf das Fehlverhalten der weniger kritischen Aufgabe nicht in der Prozesssicherheitszeit abgeschlossen werden konnte und die kritische Aufgabe T1 dadurch fehlschlägt. Das System könnte in dem Fall also nicht sicher ausgeführt werden.

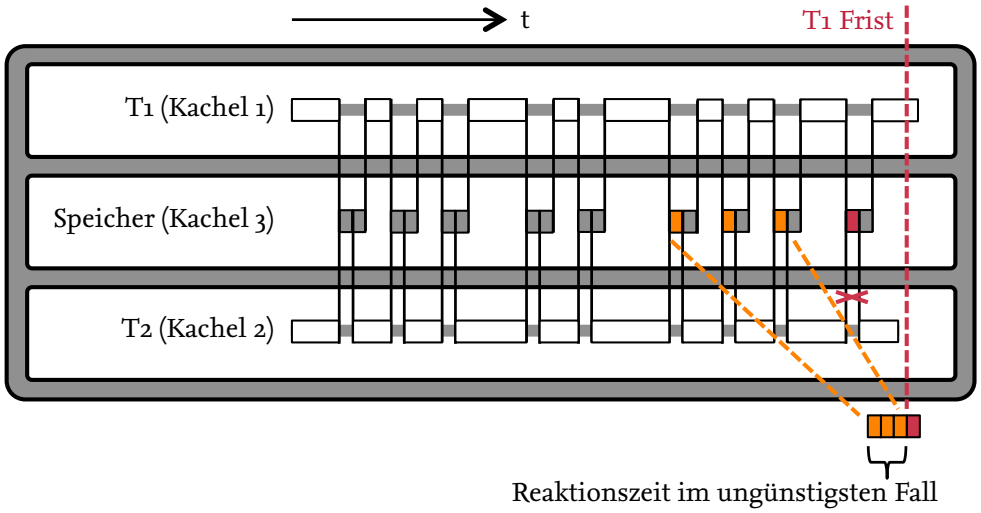

Abbildung 3.4.: Prozesssicherheitszeit (in Anlehnung an [78])

In dem dargestellten Beispiel ist die ungünstigste Situation, um die zusätzlichen fehlerhaften Zugriffe auf den gemeinsamen Speicher zu erkennen und T1 davor zu bewahren, ihre Frist zu verpassen, wenn diese zusätzlichen Zugriffe ohne Abstand zwischen den Zugriffen direkt vor der Frist von T1 erfolgen. Die Reaktionszeit für diesen ungünstigsten Fall ist in Abbildung 3.4 unten dargestellt.

Wenn zusätzliche Informationen über die Plattform verfügbar sind, wie beispielsweise, dass die weniger kritische Aufgabe T2 auf den gemeinsam verwendeten Speicher durch Verzögerungszeiten der Verbindung zwischen Prozessor und Speicher nicht schneller als zweimal pro Mikrosekunde zugreifen kann, lässt sich dieses Wissen ausnutzen, um die Reaktionszeit im ungünstigsten Fall etwas zu entschärfen.

In dem Beispiel in Abschnitt 3.1 wurden maximal 500 zusätzliche, über die erwartete Anzahl hinausgehende Zugriffe auf den gemeinsam verwendeten Speicher durch die weniger kritische Aufgabe T2 berechnet, um die Frist von T1 gerade noch garantieren zu können. Damit ergibt sich unter Ausnutzung der zusätzlichen Informationen über die Plattform eine Prozesssicherheitszeit von $\frac{500}{2/\mu s} = 250\,\mu s$. Auf einer Plattform, auf der es nicht möglich ist, zu garantieren, dass eine erhöhte Anzahl an Zugriffen auf eine gemeinsam verwendete Komponente durch eine fehlerhafte Aufgabe innerhalb der Prozesssicherheitszeit erkannt und eine angemessene Reaktion abgeschlossen werden kann, ist die gewählte Konfiguration nicht zeitlich planbar. Eine geeignete Reaktion wäre beispielsweise, wie in Abbildung 3.4 dargestellt, die Blockierung von weiteren Zugriffen der weniger kritischen Aufgabe T2 auf den gemeinsam verwendeten Speicher oder auch die komplette Deaktivierung der entsprechenden Kachel.

Kann eine geeignete Reaktion nicht innerhalb der Prozesssicherheitszeit abgeschlossen werden, kann in dem Beispiel nicht garantiert werden, dass die kritische Aufgabe T1 ihre Frist unabhängig von dem Verhalten einer möglicherweise fehlerhaften Aufgabe T2 einhalten kann. Umso länger es also dauert, ein fehlerhaftes Verhalten zu erkennen und weitere Zugriffe, Nachrichten oder energieintensive Ereignisse zu verhindern, desto weniger Aufgaben können ausreichend separiert auf einer gemeinsamen Plattform ausgeführt werden.

Eine mögliche Implementierung für die Erkennung und Unterbindung von einer gegenseitigen Beeinflussung von unterschiedlich kritischen Anwendungen könnte beispielsweise durch eine zentrale überwachende Instanz realisiert werden [36]. Auf einer Vielkernplattform müsste eine zentrale Instanz jedoch möglicherweise tausende Zähler für Zugriffe auf gemeinsam verwendete Komponenten, Abstandszähler für aufeinanderfolgende Nachrichten sowie Aktivitätszähler für energieintensive Ereignisse periodisch auslesen, die gelesenen Werte mit den erwarteten Werten vergleichen und für den Fall einer Abweichung eine angemessene Reaktion initiieren. Die Reaktionszeiten einer solchen Realisierung sind langsam und würde vor allem auch mit der Größe des Systems wachsen. Hiermit müssten mehr mögliche Fehler und größere Abweichungen in der Analyse berücksichtigt werden, was zu der bereits angedeuteten Überdimensionierung des Systems führt.

In der vorliegenden Arbeit wird daher eine in Hardware implementierte dezentralisierte Lösung vorgestellt. Eine solche Lösung, wie in Abschnitt 4.2.10 beschrieben, erlaubt es, jedes einzelne über die erwartete Anzahl hinausgehende Ereignis zu verhindern. Mit den vorgestellten Mechanismen ist daher selbst eine Prozesssicherheitszeit von Null realisierbar. Die in Abschnitt 3.1 eingeführte sichere obere Grenze für die Verzögerung $D^u_{i,S}$ durch die gemeinsame Nutzung einer Komponente mit einer weniger kritischen Aufgabe entspricht damit der erwarteten Verzögerung $D_{i,S}$ durch eine korrekt ausgeführte weniger kritische Aufgabe.

Durch die schnelle und deterministische Reaktion der in dieser Arbeit vorgestellten Laufzeitüberwachung kann also garantiert werden, dass eine weniger kritische Aufgabe selbst im Fehlerfall oder bei einem nicht spezifizierten Verhalten keinen negativen Einfluss auf eine höher kritische Aufgabe hat. Damit lässt sich die Überabschätzung der WCRT von kritischen Aufgaben reduzieren, was wiederum zu einer effizienteren Auslastung des Systems führt. Auch wenn sich die Argumentation hier auf gemeinsam verwendete Komponenten bezieht, lässt sich die Herleitung ebenfalls auf eingehende Nachrichten und den Energieverbrauch unterschiedlich kritischer Anwendungen anwenden.

3.5. Zusammenfassung

In diesem Kapitel wurde gezeigt, wie die Antwortzeit einer kritischen Anwendung durch Zugriffe auf gemeinsame Komponenten durch eine weniger oder nicht sicherheitsrelevante Anwendung verzögert werden kann und welche Auswirkung eine Begrenzung dieser Zugriffe durch eine Laufzeitüberwachung auf die Analyse hat. Weiterhin wurde in diesem Kapitel gezeigt, welchen Einfluss eingehende Interrupt-Anfragen für die Analyse einer kritischen Aufgabe haben, und existierende Ansätze zur Begrenzung des Interrupt-Aufkommens erläutert. Anschließend wurde beschrieben, warum physikalische Messung nicht für die Begrenzung des Energieverbrauchs einzelner Anwendungen zur Laufzeit eingesetzt werden können und wie die Energieabschätzung über das Zählen von Ereignissen Abhilfe schaffen kann. Aufbauend auf der Analyse des Zeitverhaltens und der Modellierung eines Systems über energieintensive Ereignisse konnte eine Analyse des Energieverbrauchs für einzelne Anwendungen und deren Beitrag zur Energiedichte in einzelnen Regionen entwickelt werden. Abschließend wurde erläutert, wie eine schnelle Reaktion einer skalierbaren Laufzeitüberwachung zu einer effizienten Ausnutzung eines Systems beitragen kann.

4 Plattformbeschreibung

In diesem Kapitel erfolgt die Vorstellung der *Integrated Dependable Architecture for Many Cores (IDAMC)*, einer konfigurierbaren, erweiterbaren und vollständig synthetisierbaren Vielkernplattform für die Erforschung von Mechanismen und Methoden zur Reduzierung des Zertifizierungsaufwandes von Mehrprozessorsystemen mit unterschiedlich kritischen Anwendungen. Konfigurierbar bedeutet hierbei, dass sie in einer generischen Hardwarebeschreibungssprache entwickelt wurde und über eine Reihe von Parametern in einer zentralen Konfigurationsdatei verfügt. Sie kann vor der Synthese auf die tatsächlichen Bedürfnisse angepasst werden und anschließend beispielsweise auf einem FPGA implementiert werden. Abbildung 4.1 zeigt eine Übersicht der Plattform, welche in [3] eingeführt und in [2] und [1] um weitere Mechanismen zur Separierung unterschiedlich kritischer Anwendungen erweitert wurde.

Bei der Entwicklung der Basisfunktionalität der IDAMC-Plattform wurde versucht, soweit wie möglich auf frei verfügbare Komponenten oder auf Komponenten mit zur Verwendung überlassenen Quellen zurückzugreifen. Daher basieren sämtliche Kacheln auf der GRLIB-Bibliothek von Aeroflex Gaisler und das NoC auf einer Vorarbeit von Kranich et al. [56]. Die im Rahmen der vorliegenden Arbeit entwickelten, neuartigen Hardwaremechanismen, unter anderem für die Virtualisierung und Separierung, befinden sich in den Netzwerkschnittstellen, welche die einzelnen Kacheln mit dem NoC verbinden. Neben der Plattform im Allgemeinen werden daher vor allem die Netzwerkschnittstellen im folgenden Kapitel im Detail beschrieben.

Der IDAMC-Plattform liegt wie bei Intels SCC und Tileras TILE64 ein Kachelkonzept zugrunde. Die einzelnen Kacheln können hier jedoch individuell konfiguriert werden und unterscheiden sich daher in der Regel voneinander. Eine Kachel kann einzelne Komponenten, wie beispielsweise einen Rechenkern, beinhalten aber auch ein komplettes busbasiertes System umfassen. Auch die Anzahl der Kacheln, die Topologie des Netzwerkes, welches die Kacheln verbindet, und die Art und Anzahl an externen Speicherschnittstellen und Peripheriemodulen ist nicht fest vorgegeben.

Abbildung 4.1 zeigt neun Knoten, die über ein zweidimensionales vermaschtes Netz (engl. *Mesh*) verbunden sind. Auch wenn das zugrunde liegende Konzept eine beliebig große Anzahl an Knoten unterstützt, lassen sich aktuell bis zu 64 Knoten auswählen. Die Knoten bestehen aus einem Switch S und bis zu vier Kacheln K (engl. *Tiles*), die über Netzwerkschnittstellen verbunden sind. Jede Kachel ist hierbei durch eine eindeutige Nummer identifizierbar. Die Kacheln basieren auf der frei verfügbaren GRLIB-Bibliothek von Aeroflex Gaisler und können wiederum komplette busbasierte Systeme mit Speicherschnittstellen, Peripheriemodulen und bis zu 16 Prozessoren sein. Hieraus ergibt sich eine theoretische Maximalanzahl an Prozessorkernen von 4096, welche aber in den meisten Fällen, aufgrund von ungenügender Chipfläche, nicht erreicht wird.

Die Plattform ist für *Asymmetric Multiprocessing (AMP)* konzipiert, was bedeutet, dass jede Kachel eine eigene Betriebssysteminstanz oder eine eigenständige Anwendung ausführt. Cache-Kohärenz in Hardware wird nicht unterstützt. Unterschiedlich kritische Anwendungen, welche eine Separierung erfordern, müssen auf Prozessorkernen in unterschiedlichen Kacheln implementiert werden, da die

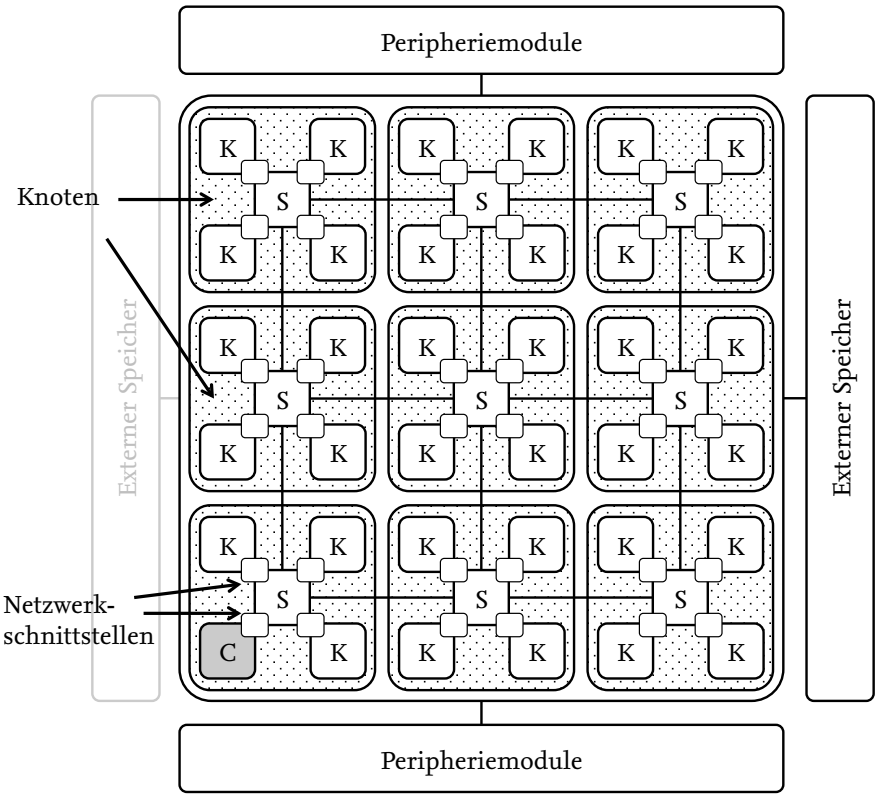

Abbildung 4.1.: Plattformübersicht

Separierungsmechanismen an den Übergängen zwischen den Kacheln und dem Netzwerk, also in den Netzwerkschnittstellen, implementiert wurden. Weiterhin bieten die Netzwerkschnittstellen Mechanismen zur Virtualisierung und flexiblen Kommunikation, um Anwendungen transparent Kacheln und verteilte Komponenten zuweisen zu können und diese Zuweisung auch zur Laufzeit ändern zu können.

Die Art und Anzahl an externen Peripherie- und Speicherschnittstellen ist ebenfalls konfigurierbar. Aktuell bietet die Plattform eine DDR2-Speicherschnittstelle, um zwei Gigabytes externen Speicher anzusprechen, eine serielle Schnittstelle, um Textausgaben an einen Steuerrechner zu senden und um von dort Befehle und Daten zu erhalten. Weiterhin lassen sich über Universalpins Funktionen mit Tastern und LEDs realisieren. Weitere Speicherschnittstellen wie Flash und SDRAM oder Peripheriemodule wie Ethernet, USB und CAN sind möglich.

Auf mindestens einer der Kacheln läuft die Systemsteuerung C. Ausschließlich von hier können Anwendungen Ressourcen zugewiesen werden, Überwachungsmaßnahmen konfiguriert sowie Systemanpassungen zur Optimierung und Fehlerbehandlung zur Laufzeit vorgenommen werden. Da die Systemsteuerung Einfluss auf sämtliche auf der Plattform laufende Anwendungen hat, auch auf sicherheitskritische Anwendungen, muss sie dieselben Anforderungen erfüllen wie die Anwendung mit der höchsten auf der Plattform vorkommenden Kritikalität.

Im Rahmen der vorliegenden Arbeit wurde die Plattform vor allem dafür verwendet, Hardwaremechanismen für die Virtualisierung und Separierung, für die Laufzeitüberwachung sowie zur sicheren kachelübergreifenden Kommunikation zu entwickeln und zu evaluieren, um so unterschiedlich kri-

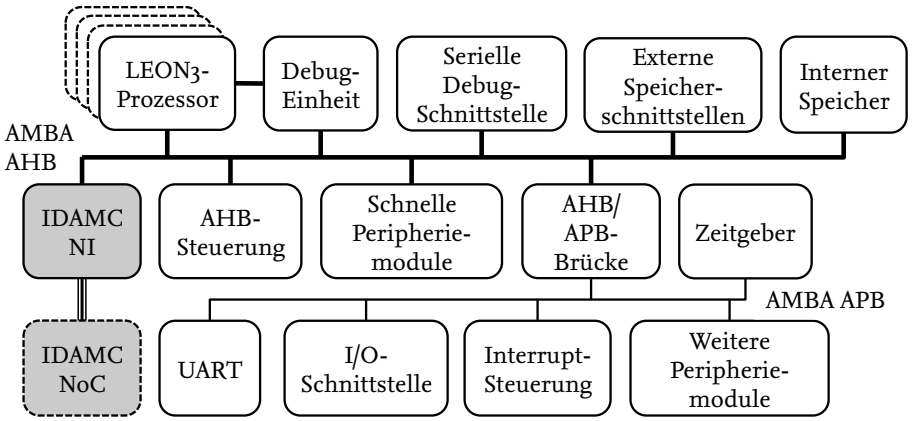

Abbildung 4.2.: Kachelübersicht

tische Anwendungen sicher, flexibel und transparent auf einer gemeinsamen Plattform integrieren zu können.

4.1. Kacheln

Für die komfortable Generierung der einzelnen Kacheln bot sich insbesondere Aeroflex Gaislers GRLIB-Bibliothek [38] an. Die Bibliothek wird vornehmlich für Forschungsplattformen und Raumfahrtanwendungen verwendet. Sie bietet eine Vielzahl an quelloffenen Komponenten, wie den LEON3 Prozessor, Interrupt- und Bus-Steuerung, Zeitgeber, Speicherschnittstellen wie SRAM, SDRAM und Flash sowie Peripheriemodule wie UART, USB, Ethernet und CAN [4]. Die Komponenten sind, wie in Abbildung 4.2 dargestellt, über AMBA-2.0 AHB- und APB-Busse [7] verbunden. Welche Komponenten Teil des Systems sind und in welcher Ausführung lässt sich über eine zentrale Konfigurationsdatei auswählen. Hierbei können auch komplette Multiprozessorsysteme mit bis zu 16 Rechenkernen konfiguriert werden. Weitere, nicht in der GRLIB-Bibliothek enthaltene Module, wie andere Prozessoren oder zusätzliche Peripherie- und Speicherschnittstellen, können zu der Bibliothek hinzugefügt werden.

Die bestehende Konfigurationsdatei der Gaisler-Bibliothek für die Generierung von busbasierten Einzel- und Mehrprozessorsystemen wurde erweitert, um mehrere auf der Gaisler-Bibliothek basierende Kacheln sowie die Anzahl der Knoten und Kacheln der IDAMC-Plattform, die Netzwerktopologie sowie weitere Parameter über eine einzelne zentrale Konfigurationsdatei festlegen zu können.

Auch wenn jede Kachel des IDAMC-Systems wiederum ein eigenes komplettes System sein kann, würde man typischerweise Kacheln als Rechenkacheln, Peripheriekacheln oder Speicherkacheln konfigurieren. Rechenkacheln umfassen dann beispielsweise einen Rechenkern, etwas lokalen Speicher und die nötigsten Peripheriemodule wie Interrupt-Steuerung und Zeitgeber. Pro Rechenkachel läuft dann eine Betriebssysteminstanz oder eine einzelne Anwendung. Speicherkacheln würden dann wiederum eine Schnittstelle zu einem externen Speicher für mehrere Rechenkacheln implementieren. Peripheriekacheln bieten dementsprechend Schnittstellen zu Peripheriemodulen.

Peripheriemodule wie USB, Ethernet oder UART in den Peripheriekacheln oder Speicherschnittstellen in den Speicherkacheln können von den Rechenkacheln gemeinsam genutzt werden. Dafür

wurde die GRLIB-Bibliothek, wie in Abbildung 4.2 dargestellt, im Rahmen der vorliegenden Arbeit um eine Netzwerkschnittstelle erweitert. Die Netzwerkschnittstelle (IDAMC NI) wird hierbei wie ein regulärer AHB-Master und AHB-Slave an den AHB-Bus angeschlossen. Als Bus-Master werden hierbei Module bezeichnet, die einen Transfer über den Bus initiieren können. Bus-Slaves dagegen können lediglich auf Anfragen von Bus-Mastern reagieren. Für die Netzwerkschnittstellen werden in den meisten Fällen beide Funktionalitäten benötigt. Für Kacheln die ausschließlich Slaves enthalten, wie beispielsweise Speicherschnittstellen, kann auf die Slave-Schnittstelle verzichtet werden.

Peripheriemodule oder Speicherschnittstellen in anderen Kacheln können über den Adressbereich der Netzwerkschnittstelle angesprochen werden. Die Netzwerkschnittstelle übersetzt Zugriffe auf die Netzwerkschnittstelle in Zugriffe auf das entsprechendes Peripheriemodul oder die entsprechende Speicherschnittstelle in anderen Kacheln. Die tatsächliche physische Lage der angesprochenen Speicher- oder Peripherieschnittstelle im System wird durch einen transparenten Übersetzungsmechanismus verborgen, so dass Komponenten in anderen Kacheln wie lokale Komponenten angesprochen werden können. Ein transparenter und flexibler Mechanismus, um mit Rechenkernen in anderen Kacheln zu kommunizieren, ist ebenfalls in den Netzwerkschnittstellen implementiert. Bestehende Anwendungen müssen daher nicht geändert werden, wenn sie auf die IDAMC-Plattform portiert werden. Auch die Entwicklung neuer Anwendungen für die Plattform wird hierdurch stark vereinfacht. Die transparente Übersetzung wird im Folgenden auch mit *Virtualisierung* bezeichnet, da sich der Mechanismus von bekannten Software-Virtualisierungslösungen zwar unterscheidet, aber dennoch Ähnlichkeiten aufweist. Ressourcen, die logisch Teil einer Kachel sind, also von lokal ausgeführten Anwendungen wie lokale Ressourcen adressierbar sind, physisch sich jedoch in einer anderen Kachel befinden, werden demnach *virtuelle Ressourcen* genannt. Aus Programmierersicht handelt es sich bei dem System um ein herkömmliches busbasiertes Einzel- oder Mehrprozessorsystem. Zuweisung von verteilten Ressourcen und eventuell auch Änderung der Zuweisung zur Laufzeit geschieht für eine lokale Anwendung transparent durch die Systemsteuerung.

4.2. Netzwerkschnittstelle

Die Netzwerkschnittstellen verbinden auf der GRLIB-Bibliothek basierende Kacheln mit dem NoC. Eine Anwendung auf einer lokalen Kachel bekommt dadurch Zugriff auf alle benötigten Ressourcen, unabhängig davon, in welchen Kacheln diese physisch implementiert sind. Abbildung 4.3 zeigt eine Übersicht einer Netzwerkschnittstelle sowie deren wichtigste Komponenten.

Details über die Lage von Komponenten, welche sich außerhalb einer lokalen Kachel befinden und möglicherweise auch zur Laufzeit wechseln können, sowie benötigte Kommunikationsverbindungen mit diesen Komponenten werden durch die Master- beziehungsweise Slave-Schnittstellen der Netzwerkschnittstellen vor lokalen Anwendungen verborgen. Sämtliche AHB-spezifischen Funktionen werden in diesen beiden Modulen gekapselt. Kacheln, die auf anderen Busprotokollen basieren, können so leicht durch Ersetzung der Master- und Slave-Schnittstelle ebenfalls eingesetzt werden.

Ein flexibler und transparenter Adressübersetzungsmechanismus innerhalb der Netzwerkschnittstellen übersetzt Zugriffe auf Komponenten, die zwar logisch, jedoch nicht physisch Teil einer lokalen Kachel sind, in Zugriffe auf die tatsächlichen physischen Komponenten. Kommunikationsverbindungen mit Rechenkernen und Peripheriemodulen in anderen Kacheln werden ebenfalls in den Netzwerkschnittstellen über einen Interrupt-Übersetzungsmechanismus für lokale Anwendungen transparent und flexibel realisiert.

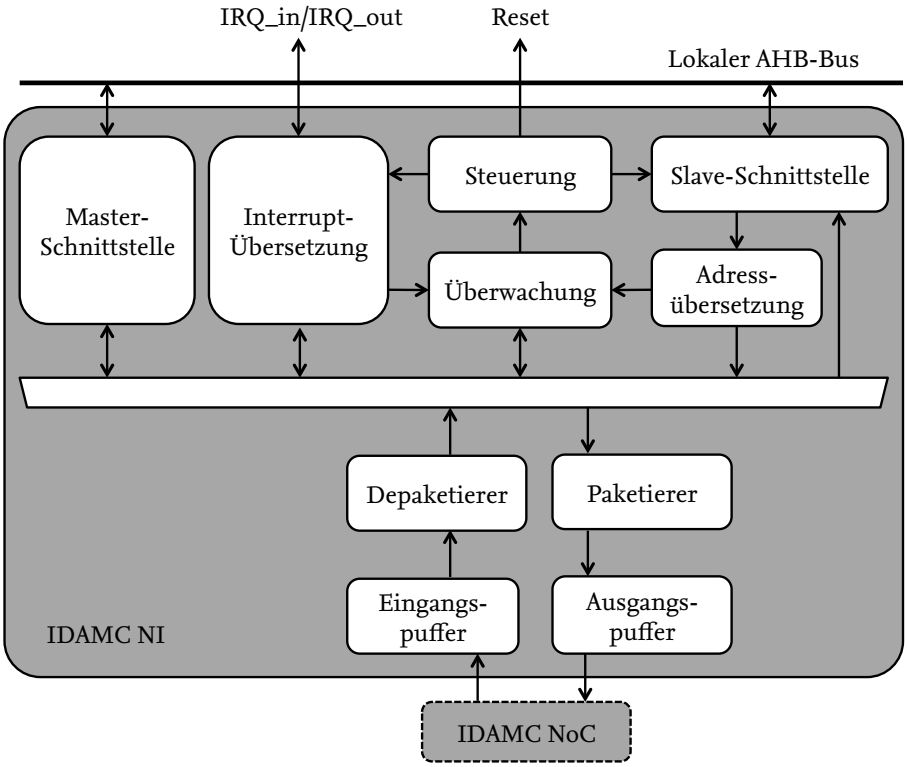

Abbildung 4.3.: Übersicht Netzwerkschnittstelle

Der Kern der vorliegenden Arbeit ist die Separierung unterschiedlich kritischer Anwendungen. Damit sich Fehler innerhalb einer Kachel nicht zu Fehlern in anderen Kacheln auswirken können, werden sämtliche von lokalen Anwendung ausgehende Aktivitäten, die zu Fehlern in anderen Kacheln führen können, in den Netzwerkschnittstellen überwacht. Geeignete Fehlerreaktionen können über die Kachelsteuerung in den Netzwerkschnittstellen entweder automatisch durch die Laufzeitüberwachung oder explizit durch die Systemsteuerung eingeleitet werden.

Um das AHB-Busprotokoll in ein paketbasiertes Netzwerkprotokoll umzusetzen und auch wieder zurück, umfassen die Netzwerkschnittstellen noch einen Paketierer/Depaketierer sowie Eingangs- und Ausgangspuffer.

4.2.1. Master-Schnittstelle

Die AHB-Masterschnittstelle wird benötigt, um lokale Peripheriemodule oder Speicherschnittstellen für Rechenkerne in anderen Kacheln zugreifbar zu machen. Die Master-Schnittstelle nimmt die Daten eines kompletten Paketes an und leitet sie entsprechend der Datenbusbreite des AHB-Busses in mehreren Teilen weiter. Leseanfragen über mehrere Wörter werden zunächst komplett gelesen und dann gesammelt an den Depaketierer weitergeleitet. Die Rückroute zum Senden der Antwort zurück an die anfragende Kachel wird hierbei aus der Hinroute berechnet.

Über die Funktion der Repräsentation von AHB-Mastern in anderen Kacheln hinaus, wird die Master-Schnittstelle auch für Zugriffe auf Komponenten innerhalb der Netzwerkschnittstelle benötigt. Interrupt- und Adressübersetzung sowie die Laufzeitüberwachung müssen bei Systemstart oder Rekonfiguration entsprechend der lokalen Anwendung programmiert werden. Die gesammelten

Daten der Laufzeitüberwachung stehen nicht nur für eine automatische dezentralisierte Auswertung innerhalb der Netzwerkschnittstelle zur Verfügung, sondern können auch manuell durch die Systemsteuerung ausgelesen werden. Ebenso lassen sich die Steuerfunktionen der Kachel manuell einsetzen. Die lokalen Funktionen und Konfigurationsmöglichkeiten der Netzwerkschnittstelle stehen jedoch aus Sicherheitsgründen ausschließlich der Systemsteuerung zur Verfügung. Lokale Anwendungen sowie Anwendungen auf anderen Kacheln, außer der Systemsteuerung, haben keinen Zugriff und die internen Funktionen der Netzwerkschnittstelle.

Die Verbindungen zwischen der Master-Schnittstelle und der Interrupt- und Adressübersetzung, der Überwachungsfunktion sowie der Kachelsteuerung sind aus Gründen der Übersichtlichkeit nicht in Abbildung 4.3 dargestellt.

4.2.2. Slave-Schnittstelle

Die Slave-Schnittstelle ist entsprechend der AMBA-2.0-Spezifikation entwickelt worden und dient dazu, die Netzwerkschnittstelle als AHB-Slave an den lokalen Bus einer Kachel anzuschließen. Die Schnittstelle ist optional und kann ausgelassen werden, falls in der lokalen Kachel entweder keine Master, in der Regel Rechenkerne, existieren oder für diese kein Zugriff auf Peripheriemodule oder Speicherschnittstellen in anderen Kacheln vorgesehen ist.

Eine der Hauptaufgaben der Slave-Schnittstelle besteht darin, die Daten eines AHB-Transfers in eine Form zu bringen, die sich leicht für den Versand über das NoC paketieren lässt und empfangene Daten in diesem Format wieder in AHB-Antworten zu transferieren. Eine für die Paketierung günstige Form liegt vor, wenn sämtliche Daten zum Versenden, auch für *Bursts*, die aus mehreren Bus-Zyklen bestehen, vor dem Versenden bereits komplett vorliegen, auch wenn hierdurch die Verzögerung eines Zugriffes auf eine Komponente in einer anderen Kacheln verlängert wird. Da das AHB-Protokoll auch vorsieht *Bursts* abzubrechen, würden Pakete, deren Versand bereits begonnen wird bevor der *Burst* abgeschlossen ist, nicht komplett versendet werden können, was zu einem inkonsistenten Netzwerkzustand führen kann.

Da Kacheln zum einen Rechenkerne enthalten können, die Peripheriemodule oder Speicherschnittstellen in anderen Kacheln nutzen, und zum anderen selbst Peripheriemodule und Speicherschnittstellen für Rechenkerne in anderen Kacheln zur Verfügung stellen können, implementiert die Slave-Schnittstelle auch die in der AMBA-2.0-Spezifikation vorgesehenen optionalen *Split*-Übertragungen. *Split*-Übertragungen erlauben es, angefangene Anfragen an einen Slave, deren Abschluss nicht in kurzer Zeit zu erwarten ist, zu unterbrechen, den Bus für andere Aktionen freizugeben und den Transfer später fortzusetzen [7], sobald die entsprechende Antwort verfügbar ist. Diese optionale Funktionalität wurde in den Netzwerkschnittstellen implementiert, da Anfragen an einen Slave in einer weit entfernte Kachel zu teils erheblichen Verzögerungen führen können. Würde der Bus für diese Zeit ohne *Split*-Übertragungen blockiert werden, würden damit sämtliche lokale Anwendungen und Anwendungen in anderen Kacheln, welche Peripheriemodule und Speicherschnittstellen in derselben Kachel verwenden, ebenfalls für diese Zeit blockiert werden. *Split*-Übertragungen erlauben Anwendungen auf anderen Rechenkernen fortzufahren, ohne auf den blockierten Rechenkern, der die Anfrage an die entfernte Komponente gestellt hat, warten zu müssen.

Neben den lokalen Adressbereichen, in die virtuelle Speicherschnittstellen und Peripheriemodule aus anderen Kacheln eingeblendet werden können und die im Folgenden kurz erläutert werden, stellt die Slave-Schnittstelle noch einen weiteren kleinen Adressbereich lesend für lokale Anwendungen

zur Verfügung. Hierüber können Konfigurationsdaten oder beispielsweise auch die Kachelidentifikationsnummer ausgelesen werden.

Adressbereiche

Bis zu vier unterschiedliche Adressbereiche können laut [38] an AHB-Slave-Schnittstellen zugewiesen werden. Mögliche Typen sind hier AHB-Speicherbereiche, AHB-I/O-Bereiche und APB-I/O-Bereiche. Drei der Adressbereiche werden für die Netzwerkschnittstelle der IDAMC-Plattform verwendet. Einmal als Konfigurationsbereich der Netzwerkschnittstelle selbst, einmal für Speicherschnittstellen oder komplexe Peripheriemodule in anderen Kacheln und einmal für einfache Peripheriemodule oder Allzweckeingaben und -ausgaben (engl. *General-Purpose Input/Output (GPIO)*) in anderen Kacheln. Tabelle 4.1 zeigt ein Beispiel für eine mögliche Anordnung der Adressbereiche. Die Endung $\_MADDR$ in $NETIF\_LOC\_MADDR$ und $NETIF\_MEM\_MADDR$ identifiziert einen Adressbereich in der Größe eines Vielfachen von einem Megabyte und wird hauptsächlich für größere Adressbereiche für Speicherschnittstellen oder komplexere Peripheriemodule verwendet. Diese Adressbereiche werden in der GRLIB-Bibliothek als AHB-Speicherbereiche bezeichnet. Die Endung $\_IADDR$ in IO_IADDR spezifiziert einen Adressbereich in der Größe eines Vielfachen von 256 Bytes und wird vor allem für kleinere Adressbereiche wie einfache Peripheriemodule oder zur Ansteuerung von Allzweckeingaben und -ausgaben verwendet. Diese Bereiche werden in der GRLIB-Bibliothek als AHB-I/O-Bereiche bezeichnet. Größere Adressbereiche als ein Megabyte beziehungsweise 256 Bytes können über Masken spezifiziert werden. Die Maske $NETIF\_MEM\_MMASK$ gehört zu $NETIF\_MEM\_MADDR$ und die Maske $NETIF\_IO\_IMASK$ zu $NETIF\_IO\_IADDR$. Der Konfigurationsbereich der Netzwerkschnittstelle, welcher sich an der Adresse $NETIF\_LOC\_MADDR$ befindet, ist immer einen Megabyte groß, weshalb hier auf eine Maske verzichtet wird. Die negierten Bits der Maske ergeben durch eine Oder-Verknüpfung mit der Basisadresse die Endadresse des spezifizierten Adressbereiches, das heißt, an der Stelle jeder Null in der Maske hat die Endadresse eine Eins. Damit wird beispielsweise aus der Basisadresse $NETIF\_MEM\_MADDR = 0x400$ und der Maske $NETIF\_MEM\_MMASK = 0xFF0$ der Adressbereich $0x40000000 - 0x40FFFFFF$. Die Startadressen der benötigten Adressbereiche sowie die zugehörigen Masken können in der zentralen Konfigurationsdatei für jede Kachel individuell festgelegt werden.

Konfigurationsbereich der Netzwerkschnittstelle Der lokale Konfigurationsbereich der Netzwerkschnittstelle beginnt bei der Adresse $NETIF\_LOC\_MADDR$ und benötigt einen Megabyte an Adressen. Die Adresse $NETIF\_LOC\_MADDR$ ist für alle Kacheln gleich, damit eine lokale Anwendung, unabhängig von der zugeordneten Kachel, den Konfigurationsbereich stets an derselben Stelle findet, um so ohne Softwareanpassungen auf beliebige Kacheln platziert werden zu können. Die Adresse $NETIF\_LOC\_MADDR$ wird mit den zwölf signifikantesten Bits der lokalen Adresse auf dem AHB-Bus verglichen. Dies entspricht den Bits 31 bis 20. Von einer Anwendung, die lokal auf der Kachel ausgeführt wird, darf jedoch nur lesend auf den Konfigurationsbereich der Netzwerkschnittstelle zugegriffen werden. Schreiben darf nur die Systemsteuerung, also die Kachel mit der Identifikationsnummer 0. Diese darf wiederum auch ihren eigenen Konfigurationsbereich selbst programmieren. Um die Netzwerkschnittstelle einer Kachel zu konfigurieren, blendet die Systemsteuerung den Konfigurationsbereich der entfernten Kachel in den Speicheradressbereich ihrer eigenen Netzwerkschnittstelle ein und schreibt dann auf diesen wie auf ein lokales Modul. Die

31 20	19 8	7 0	Beschreibung
0x000 - NETIF_LOC_MADDR -1	0x000 - 0xFFF	0x00 - 0xFF	Andere Module
NETIF_LOC_MADDR*	0x000 - 0xFFF	0x00 - 0xFF	Netzwerkschnittstelle Konfiguration
NETIF_MEM_MADDR* - NETIF_MEM_MADDR OR NOT(NETIF_MEM_MMASK)	0x000 - 0xFFF	0x00 - 0xFF	Entfernte Speicherbereiche
AHBIO	NETIF_IO_IADDR* - NETIF_IO_IADDR OR NOT(NETIF_IO_IMASK)	0x00 - 0xFF	Entfernte I/O-Bereiche
AHBIO +1 - 0xFFF	0x000 - 0xFFF	0x00 - 0xFF	Andere Module

*die Adressbereiche der Netzwerkschnittstelle müssen nicht notwendigerweise aufeinanderfolgend oder ohne Lücken sein

Tabelle 4.1.: Beispieladressbereiche der Netzwerkschnittstelle

einzelnen Submodule im Konfigurationsbereich der Netzwerkschnittstelle werden über die Bits 19 bis 17 angesprochen. Die entsprechende Codierung ist in Tabelle 4.2 dargestellt.

31 20	19 17	16 0	Beschreibung
NETIF_LOC_MADDR	0x0	0x-	Reserviert
	0x1	0x-	Interrupt-Übersetzung
	0x2	0x-	Interrupt-Überwachung
	0x3	0x-	Adressübersetzung
	0x4	0x-	Kachelsteuerung
	0x5	0x-	Reserviert
	0x6	0x-	Reserviert
	0x7	0x-	Adressüberwachung

Tabelle 4.2.: Submodule Adressierung

Entfernte Speicherbereiche Der lokale Adressbereich der Netzwerkschnittstelle, welcher zum Ansprechen größerer entfernter Adressbereiche dient, beginnt bei der konfigurierbaren Startadresse *NETIF_MEM_MADDR*. Dieser Bereich wird verwendet, um zum Beispiel größere Adressbereiche für Speicherschnittstellen oder komplexe Peripheriemodule, welche einen Adressbereich von mindestens einem Megabyte benötigen, in der lokalen Kachel einzublenden. *NETIF_MEM_MADDR* wird hierbei mit den zwölf signifikantesten Bits des AHB-Adressbusses verglichen, was bei einem 32-Bit Adressbus den Bits 31 bis 20 entspricht. Die Bits 19 bis 0 werden bei der Auswahl des Adressbereiches für entfernte Speicherbereiche nicht betrachten, sondern werden unverändert an das tatsächlich angesprochene Ziel weitergeleitet.

Entfernte I/O-Bereiche Der I/O-Adressbereich der Netzwerkschnittstelle dient dazu, kleinere Adressbereiche für einfache Peripheriemodule oder einfache Eingabe- und Ausgabefunktionen anzusprechen. Die konfigurierbare Startadresse beginnt bei $NETIF\_IO\_IADDR$. Jeder Block umfasst hierbei 256 Bytes. Die Bits von $NETIF\_IO\_IADDR$ werden hier mit den Bits 19 bis 8 des AHB-Adressbusses verglichen. Die Bits 7 bis 0 werden im tatsächlich angesprochenen Adressbereich in der Zielkachel dekodiert. Die höheren Bits 31 bis 20 sind für alle I/O-Adressbereiche gleich. Der Parameter $AHBIO$ definiert einen Megabyte an Adressen, welcher zwischen sämtlichen I/O-Adressbereichen, also den der Netzwerkschnittstelle und anderer Module, aufgeteilt wird.

4.2.3. Paketierer

Der Paketierer setzt aus Daten und zusätzlichen Kontrollinformationen von Master- und Slave-Schnittstelle, Laufzeitüberwachung sowie Interrupt-Übersetzung Pakete zusammen, die über das NoC versendet werden können. Die Daten und Kontrollinformationen, wie Pakettyp, Route, Kanal und Kachelidentifikationsnummer müssen zu demselben Zeitpunkt komplett anliegen. Der Pakettyp identifiziert das absendende Modul, die Route und der Kanal werden von der Adress- oder Interrupt-Übersetzung vorgegeben und die Kachelidentifikationsnummer dient dazu, Zugriffsbeschränkungen umsetzen zu können. Uneingeschränkter Zugriff zu allen Kacheln und Konfigurationsmöglichkeiten wird nur der Systemsteuerung mit der Kachelidentifikationsnummer 0 gewährt.

Wenn der Ausgangspuffer freie Kapazitäten für den gewählten Kanal signalisiert, werden die zu sendenden Kontroll- und Dateninformationen in *Physical Digits (Phits)* umgewandelt und an den Ausgangspuffer weitergeleitet. Vier solcher *Phits* bilden ein *Flow Control Digit (Flit)* und werden immer direkt nacheinander gesendet. Nach dem Weiterleiten eines *Flits* wird, falls weitere *Flits* gesendet werden sollen, zunächst wieder auf freie Kapazitäten geprüft. Der Paketierer fügt abhängig von der zu sendenden Anzahl an Daten den entsprechenden Typ des *Flits* (*Head*, *Body*, *Tail* oder Einzel-*Flit*) automatisch ein. Weitere Informationen zum Paketformat und dem NoC im Allgemeinen können in Abschnitt 4.3 nachgelesen werden.

4.2.4. Depaketierer

Der Depaketierer ist das Gegenstück zum Paketierer. Er nimmt verfügbare Daten im Eingangspuffer *Phit* für *Phit* an und stellt Daten- und Kontrollinformationen an den Ausgängen wieder im geforderten Format dar. Ein Paket muss vollständig empfangen sein bevor die Ausgänge des Paketierers als gültig markiert werden. Das Ende des Pakets ist durch den *Flit*-Typ *Tail* markiert.

Abhängig vom empfangenen Pakettyp werden die Daten entweder an die Master-Schnittstelle als Anfrage oder an die Slave-Schnittstelle, Interrupt-Übersetzung oder die Überwachungseinheit als Antwort weitergeleitet.

4.2.5. Ausgangspuffer

In der aktuellen Implementierung werden Daten nicht, wie der Name des Moduls andeutet, zwischengespeichert. Die Hauptaufgabe des Ausgangspuffers liegt derzeit darin, den verfügbaren Platz in den Eingangspuffern des angeschlossenen Switches durch einen Zähler pro virtuellem Kanal zu registrieren. Jedes gesendete *Flit* verringert den Zähler und jedes durch den angeschlossenen Switch weitergeleitete *Flit* erhöht den entsprechenden Zähler.

Für den Fall, dass Daten deutlich schneller in der Kachel versendet werden sollen als das NoC diese weiterleiten kann, könnte der Ausgangspuffer auch als tatsächlicher Puffer ausgeführt werden. Für

sehr große System, deren korrektes Zeitverhalten bei der Implementierung schwierig zu realisieren ist, könnten die Puffer auch dazu verwendet werden, die Zeitbasis der Kacheln von der Zeitbasis des NoCs zu entkoppeln und so die IDAMC-Plattform als *GALS*-System (engl. *Globally Asynchronous Locally Synchronous*) zu implementieren.

4.2.6. Eingangspuffer

Der Eingangspuffer der Netzwerkschnittstelle ist an den Eingangspuffern der Switches angelehnt entworfen worden (siehe dazu Abschnitt 4.3). Der Eingangspuffer speichert bis zu fünf *Flits* pro virtuellem Kanal in FIFO-Puffern (engl. *First-In First-Out*), was 20 *Phits* entspricht, um so Ressourcen des NoCs möglichst schnell wieder freigeben zu können. Jedes vierte *Phit* wird automatisch überprüft, um den zugehörigen Kanal des aktuellen *Flits* zu bestimmen, um es so in den korrekten FIFO-Puffer einsortieren zu können. Der Füllstand der FIFO-Puffer wird ständig überprüft, um die Verfügbarkeit eines neuen *Flits* nach Kanälen getrennt an den Depaketierer weiterleiten zu können. Enthalten mehrere FIFO-Puffer ein neues *Flit*, wird der nächste aktive Puffer nach dem *Round-Robin*-Verfahren ermittelt, bei dem die Puffer in einer zuvor festgelegten Abfolge der Reihe nach abgearbeitet werden. Nach der erfolgreichen Weiterleitung eines *Flits* an den Depaketierer wird der freigewordene Speicherplatz an den angeschlossenen Switch gemeldet.

4.2.7. Kachelsteuerung

Jede Kachel verfügt in der Netzwerkschnittstelle über eine Kachelsteuerung, die zum einen für automatische Reaktionen der Überwachungsfunktionen als auch explizit durch die Systemsteuerung verwendet werden kann. Hierüber lässt sich eine Kachel unabhängig von den anderen Kacheln zurücksetzen oder auch über dauerhaftes Zurücksetzen komplett deaktivieren. Bei Systemstart sind alle Kacheln außer Kachel 0, welche die Systemsteuerung darstellt, deaktiviert und müssen explizit durch die Systemsteuerung aktiviert werden. Die Kachelsteuerung kann in Verbindung mit den Überwachungsfunktionen neben dem Zurücksetzen und Deaktivieren der Kachel auch zur Verzögerung von ausgehenden Interrupt-Anfragen sowie ausgehenden Zugriffen auf Speicherschnittstellen und Peripheriemodulen in anderen Kacheln verwendet werden.

Die Skalierung beziehungsweise die Abschaltung der Taktfrequenz (engl. *Frequency Scaling/Gating*) oder auch die Abschaltung der Spannungsversorgung (engl. *Power Gating*) für einzelne Regionen des Chips könnte zur Reduzierung der Verlustleistung eingesetzt werden. Da in aktuellen FPGAs die Skalierung oder Abschaltung der Frequenz oder die Abschaltung der Spannungsversorgung für einzelne Regionen des Chips gar nicht oder nicht effizient umgesetzt werden kann, wurden diese Techniken hier nicht implementiert, auch wenn die Kachelsteuerung hierfür der richtige Ort wäre. Des Weiteren gehörte nicht die Reduzierung der Verlustleitung zu den Hauptzielen bei der Entwicklung der IDAMC-Plattform, sondern die sichere Separierung der Verlustleistung von unterschiedlich kritischen Anwendungen.

4.2.8. Adressübersetzung

Die Adressübersetzung in den Netzwerkschnittstellen bildet den relativ kleinen Adressbereich einer einzelnen Kachel von vier Gigabytes in den möglicherweise deutlich größeren Adressbereich des Gesamtsystems ab. Dieser ist theoretisch unbegrenzt und wird durch die Anzahl der Kacheln des Systems definiert. Die Adressübersetzung ist für lokale Anwendungen transparent, was bedeutet, dass die tatsächlichen physischen Ziele nicht bekannt sein müssen. Dies erleichtert, wie bereits

erläutert, die Verwendung von bestehenden Einzel- und Mehrprozessoranwendungen auf der hier beschriebenen Vielkernplattform. Darüber hinaus können neue Anwendungen unabhängig von anderen Anwendungen, die später auf derselben Plattform gemeinsam laufen sollen, entwickelt werden. Welche Komponenten zur Laufzeit tatsächlich genutzt werden und wie diese zwischen mehreren Anwendungen aufgeteilt oder von diesen gemeinsam verwendet werden, wird zentral von der Systemsteuerung verwaltet und vor den einzelnen Anwendungen verborgen.

Die Adressübersetzung ersetzt hierbei nicht die mögliche Speicherverwaltungseinheit (engl. *Memory Management Unit (MMU)*) der LEON3-Prozessoren. Diese können weiterhin durch eine entsprechende Konfiguration der Prozessoren verwendet werden, um lokale virtuelle Adressen in lokale physische Adressen umzusetzen. Die Adressübersetzung übersetzt dann wiederum den lokalen physischen Adressbereich der Netzwerkschnittstelle in physische Adressen in beliebigen Kacheln des Gesamtsystems. Physische Speicherschnittstellen oder Peripheriemodule in anderen Kacheln können so logisch in den lokalen Adressraum eingeblendet werden. Die Adressübersetzung ist daher ein wichtiger Bestandteil des in dieser Arbeit verwendeten Begriffs der *Hardware-Virtualisierung*, welcher in Abschnitt 4.5 noch einmal genauer erläutert wird.

Angelehnt an die in der GRLIB-Bibliothek eingeführten Adressbereiche für Speicherbereiche und I/O-Bereiche basiert die Adressübersetzung auf zwei logischen Tabellen, eine Tabelle für Speicheradressbereiche, also Adressblöcke von einem Megabyte, und eine Tabelle für I/O-Adressbereiche, also Adressblöcke von jeweils 256 Bytes. Um verfügbare Ressourcen des FPGAs zu schonen, sind beide Tabellen jedoch in einem gemeinsamen BRAM implementiert.

Jede der Tabellen enthält einen Eintrag pro adressierbarem Adressenblock. Die Lage der Tabellen sowie die Anzahl der jeweiligen Einträge wird vor der Implementierung, wie in Abschnitt 4.2.2 erläutert, durch die Parameter *NETIF_MEM_MADDR* und *NETIF_MEM_MMASK* beziehungsweise *NETIF_IO_IADDR* und *NETIF_IO_IMASK* festgelegt, wodurch auch die Größe der Tabellen definiert wird. Der entsprechende Tabelleneintrag für ein anzusprechendes Modul wird über dessen lokale Basisadresse ausgewählt. Für Speicheradressbereiche sind dies die Bits 31 bis 20 der lokalen AHB-Adressen und für I/O-Adressbereiche die Bits 19 bis 8, wie ebenfalls in Abschnitt 4.2.2 beschrieben wird. Für I/O-Adressbereiche liegen die oberen zwölf Bits, also die Bits 31 bis 20, fest auf dem Wert des konfigurierbaren Parameters *AHBIO*. Die übrigen Bits, der Offset, also die Bits 19 bis 0 für Speicheradressbereiche und die Bits 7 bis 0 für I/O-Adressbereiche, werden unverändert an das angesprochene Modul in der Zielkachel weitergeleitet.

Jeder Tabelleneintrag enthält, wie in Abbildung 4.4 für Speicheradressbereiche dargestellt, die Route und den virtuellen Kanal, um den Zugriff als Paket über das NoC zu der entsprechenden Zielkachel zu senden, die Basisadresse der angesprochenen Komponente in der Zielkachel, welche die Basisadresse in der lokalen Kachel ersetzt, sowie Schutzbits, welche für schreib- oder für lesegeschützte Bereiche verwendet werden können. Die hier verwendete tabellenbasierte Festlegung der Route für ausgehende Pakete erleichtert die flexible Umleitung von Paketen für Optimierungen oder Fehlerbehandlungen. Die Tabelle für I/O-Adressbereiche sieht ähnlich aus. Sie unterscheidet sich lediglich in der für die Adressierung verwendeten Basisadresse.

Speicherschnittstellen und Peripheriemodule in anderen Kacheln, welche nicht zuvor durch die Systemsteuerung einer der beiden Tabellen hinzugefügt wurden, können nicht durch eine lokale Anwendung angesprochen werden. Hiermit kann eine räumliche Separierung von unterschiedlich kritischen Anwendungen erreicht werden. Die Adressbereiche von unterschiedlichen Kacheln kön-

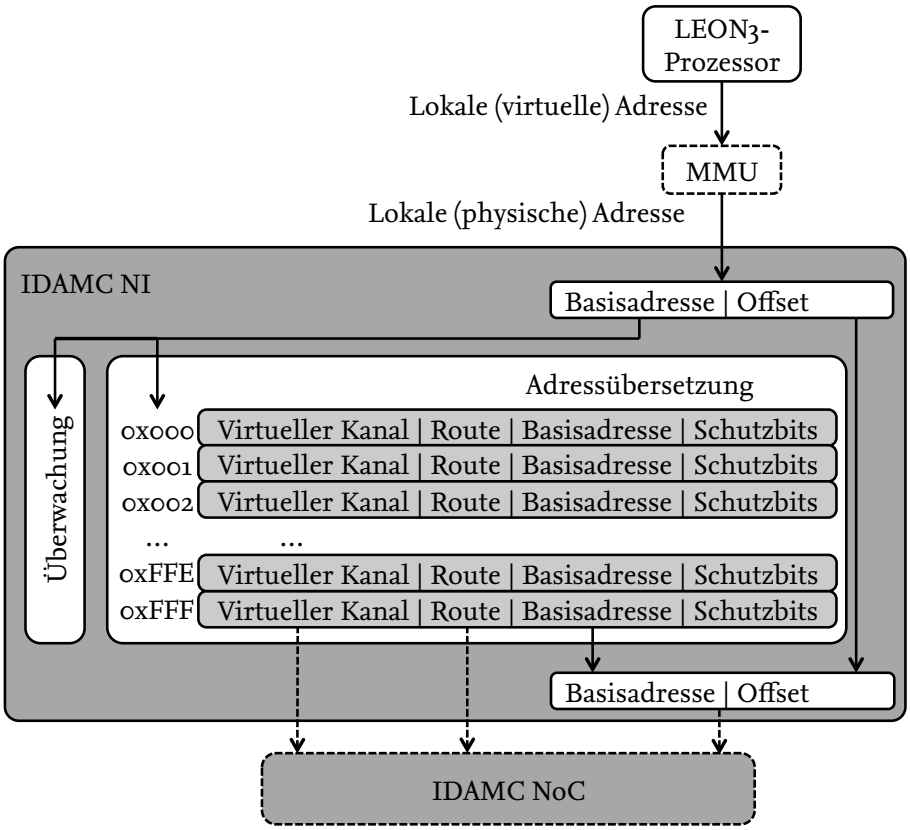

Abbildung 4.4.: Adressübersetzung

nen disjunkt oder überlappend konfiguriert werden, je nachdem ob sie exklusiv benutzt werden sollen, um die Adressbereiche der einzelnen Anwendungen räumlich voneinander zu trennen, oder ob sie für eine sichere Interaktion zwischen Anwendungen eingesetzt werden sollen.

Die Tabelle kann zur Laufzeit umprogrammiert werden, um beispielsweise fehlerhafte Ziele auszutauschen. Hierüber kann die Verfügbarkeit der lokalen Anwendungen erhöht werden. Die Programmierung der Tabelle kann jedoch nur von der Systemsteuerung durchgeführt werden. Lokale Anwendungen und Anwendungen auf anderen Kacheln haben keinen Zugriff. Die Neuprogrammierung kann weiterhin dafür verwendet werden, um Anwendungen bei einem nicht spezifiziertem Verhalten weiteren Zugriff auf bestimmte Komponenten zu verwehren. So kann verhindert werden, dass sich Fehler in einer lokalen Anwendung auf das restliche System auswirken können. Die Zugriffe auf die jeweiligen Adressbereiche können hierfür, wie in Abbildung 4.4 angedeutet und in Abschnitt 4.2.10 genauer erläutert, durch die Laufzeitüberwachung in den Netzwerkschnittstellen überwacht werden.

Durch die flexible und transparente Programmierung der Adressübersetzungstabellen lassen sich Anwendungen auch ohne hierfür vorgesehen zu sein auf andere Kacheln migrieren. Hierfür müssen die Adressübersetzungstabellen der Zielkachel lediglich mit den verteilten Komponenten programmiert werden, welche von der zu migrierenden Anwendung verwendet werden. Anschließend kann die Anwendung auf der Zielkachel neu gestartet werden.

4.2.9. Interrupt-Übersetzung

Jede Rechenkachel verfügt über eine lokale Interrupt-Steuerung, welche in der GRLIB-Bibliothek von Aeroflex Gaisler enthalten ist. Die Multiprozessor-Interrupt-Steuerung IRQMP [4] unterstützt bis zu 16 angeschlossene LEON3-Prozessoren, 15 unterschiedliche maskierbare Interrupt-Nummern sowie zwei unterschiedliche Interrupt-Klassen. Jede Interrupt-Nummer kann explizit durch Anwendungen auf den angeschlossenen Prozessoren durch Schreibzugriffe auf die Interrupt-Steuerung gesetzt werden. Hierdurch lässt sich über das Setzen von Interrupt-Nummern (engl. *Interrupt Request Level (IRL)*) für andere angeschlossene Prozessoren eine Kommunikation zwischen Prozessoren aufsetzen.

Von den aktiven Interrupts leitet die Interrupt-Steuerung den Interrupt mit der höchsten Priorität an alle angeschlossenen Prozessoren weiter, sofern dieser nicht zuvor von den jeweiligen Prozessoren maskiert wurde. Die Priorität ist um so höher, je höher die Interrupt-Klasse und je höher die Interrupt-Nummer ist. Der Interrupt in der Interrupt-Klasse eins mit der Interrupt-Nummer 15 hat daher die höchsten Priorität. Ein empfangener Interrupt wird von den Prozessoren automatisch bestätigt.

In der GRLIB-Bibliothek wurden 32 Interrupt-Leitungen zu den AHB- und APB-Bussen hinzugefügt. Von dem IRQMP-Modul werden jedoch nur die Leitungen 15 bis eins verwendet. Die Leitungen können von allen angeschlossenen Komponenten unabhängig voneinander gelesen und auch gesetzt werden. Interrupts können dadurch auch geteilt werden. Wird ein Interrupt an einen Prozessor weitergeleitet, muss dieser bei einer geteilten Interrupt-Nummer zunächst alle infrage kommenden Interrupt-Quellen überprüfen, um die auslösende Komponente zu ermitteln.

Die lokale Interrupt-Steuerung IRQMP aus der Gaisler-Bibliothek wird in den Rechenkacheln für die Kommunikation zwischen lokalen Prozessoren und für die Benachrichtigung von Prozessoren durch angeschlossene lokale Peripheriemodule verwendet. Für die Kommunikation zwischen Prozessoren in unterschiedlichen Kacheln und für die Benachrichtigung eines Prozessors in einer anderen Kachel durch ein lokales Peripheriemodul verfügt die Netzwerkschnittstelle über ein Mechanismus zur Interrupt-Übersetzung, welcher zusammen mit Martens [61] entwickelt wurde.

Eine Anwendung, die sich flexibel und transparent zu verfügbaren Ressourcen einer Vielkernplattform zuweisen lässt und deren Zuordnung sich auch zur Laufzeit ändern kann, erfordert es, dass auch ihre Kommunikationsverbindungen mit verwendeten Peripheriemodulen und anderen Prozessoren flexibel und transparent sind. Damit sich die Interrupt-Behandlung immer gleich verhält, unabhängig davon, ob sich Interrupt-Quelle und Interrupt-Ziel in derselben oder in unterschiedlichen Kacheln befinden, orientiert sich die Interrupt-Übersetzung in den Netzwerkschnittstellen an der Interrupt-Behandlung der GRLIB-Bibliothek.

Wenn für die Interrupt-Übersetzung, entsprechend der weiter oben beschriebenen Adressübersetzung, anstatt physischer Interrupt-Quellen und –Ziele, virtuelle Interrupt-Quellen und -Ziele verwendet werden, lassen sich Anwendungen, die mit Komponenten außerhalb der lokalen Kachel kommunizieren, ohne Software-Änderungen beliebigen Kernen und Peripheriemodulen zuordnen und diese Zuordnung zur Laufzeit ändern. Da hiermit eine Anwendung keine Kenntnisse darüber benötigt, wo sich eine bestimmte Interrupt-Quelle oder ein bestimmtes Interrupt-Ziel auf der Vielkernplattform befindet, erleichtert dies die verteilte Integration von Anwendungen, die ursprünglich für busbasierte Einzel- oder Mehrprozessorsysteme entworfen wurden. Der im folgenden beschriebene Interrupt-Übersetzungsmechanismus ist, neben der Adressübersetzung, eine

weitere Voraussetzung dafür, dass sich die Zuordnung von Anwendungen zu verteilten Ressourcen auch zur Laufzeit flexibel und transparent für Optimierungen und Fehlerbehandlungen ändern lässt.

Die Transparenz und Flexibilität ist, neben der bloßen Realisierung von kachelübergreifenden Interrupt-Anfragen, ein weiterer Vorteil gegenüber der existierenden Interrupt-Steuerung aus der Bibliothek von Aeroflex Gaisler. Indem ein weiteres redundantes Ziel zunächst maskiert lokal vorgehalten wird und bei Bedarf demaskiert wird, wobei gleichzeitig das ursprüngliche Ziel deaktiviert wird, könnte das IRQMP-Modul zwar ebenfalls dafür eingesetzt werden, ein lokales Interrupt-Ziel zur Laufzeit zu ersetzen, die transparente und flexible Ersetzung von (fehlerhaften) Interrupt-Quellen ohne Anpassung der Software ist jedoch nicht mit dem Modul der Gaisler-Bibliothek möglich. Laufende Anwendungen erwarten die Quelle für einen aktiven Interrupt immer an derselben Adresse, welche mit der herkömmlichen Variante nicht auf die Adresse einer anderen Interrupt-Quelle übersetzt werden kann.

Die Zuordnung von virtuellen zu physischen Interrupt-Quellen und –Zielen in anderen Kacheln ist durch den Interrupt-Übersetzungsmechanismus implementiert, welcher in Abbildung 4.5 zu sehen ist. Interrupts zwischen lokalen Rechenkernen und lokalen Peripheriemodulen werden weiterhin von der Interrupt-Steuerung aus der Gaisler-Bibliothek behandelt. Die Netzwerkschnittstelle ist ebenfalls mit der lokalen Interrupt-Steuerung verbunden und repräsentiert dabei alle Interrupt-Ziele und Interrupt-Quellen, die nur logisch zu der lokalen Kachel gehören, sich physisch jedoch in anderen Kacheln befinden. Im Folgenden werden Komponenten, die logisch, aber nicht physisch Teil einer Kachel sind, virtuelle Prozessoren und virtuelle Peripheriemodule genannt. In Abbildung 4.5 werden diese mit vProzessor und vPeripherie gekennzeichnet. Für eine lokale Anwendung ist es nicht zu erkennen, dass diese Komponenten sich nicht in derselben Kachel befinden.

Insgesamt können, wie bereits erläutert und in Abbildung 4.5 dargestellt, bis zu 16 Prozessoren über die Interrupt-Leitungen *IRQ_Out* an die Interrupt-Steuerung aus der GRLIB-Bibliothek von Aeroflex Gaisler angeschlossen werden. Vor der Implementierung kann die Anzahl der lokalen physischen Kerne 0 bis $m - 1$ sowie die lokal verwendeten Prozessoren aus anderen Kacheln m bis $n - 1$ mit $n \leq 16$ über die zentrale Konfigurationsdatei festgelegt werden. Ein Interrupt für eine Komponente außerhalb derselben Kachel wird, wie für einen lokalen Prozessor auch, aktiviert und von der lokalen Interrupt-Steuerung behandelt. Die Maskierung muss dafür zunächst von der Systemsteuerung für die entsprechende Interrupt-Nummer entfernt werden. Anschließend wird dieser Interrupt an die Netzwerkschnittstelle weitergeleitet und von dort an die zuvor festgelegte Kachel gesendet. Entsprechend der Interrupt-Behandlung in der GRLIB-Bibliothek werden Interrupt-Anfragen, welche an die Netzwerkschnittstelle weitergeleitet werden, so wie Interrupts an lokale Prozessoren direkt automatisch bestätigt.

Ankommende Interrupts werden durch das vPeripherie-Modul repräsentiert. Das Modul nimmt empfangene Interrupt-Anfragen von dem Depaketierer entgegen und setzt die entsprechenden lokale Interrupt-Leitungen *IRQ_In*, welche durch eine Oder-Verknüpfung auch mit den Interrupt-Ausgängen der lokalen Peripheriemodule verbunden sind. Anschließend wird der Interrupt von der lokalen Interrupt-Steuerung genauso wie eine Interrupt-Anfrage einer lokalen Komponente behandelt.

Die Übersetzung von lokalen Interrupts für virtuelle Rechenkerne geschieht, ähnlich wie bei der Adressübersetzung, über eine Tabelle. Die Tabelle für die Interrupt-Übersetzung enthält, wie in

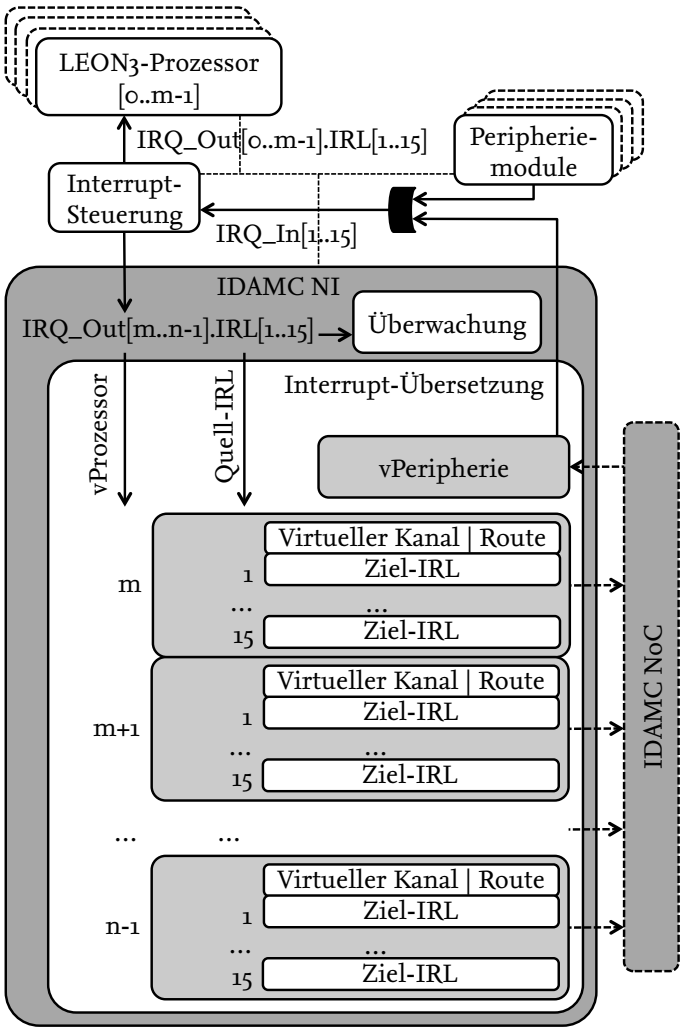

Abbildung 4.5.: Interrupt-Übersetzung

Abbildung 4.5 zu sehen, einen Satz an Einträgen für jeden virtuellen Prozessor. Jeder Satz wiederum erhält die zu verwendende Route und den virtuellen Kanal zu dem repräsentierten physischen Prozessor. Außerdem sind die Interrupt-Nummern in der Zielkachel (Ziel-IRL) enthalten, welche nicht zwangsläufig dieselben sein müssen wie in der Quellkachel. Auch diese Tabelle kann ausschließlich von der Systemsteuerung programmiert werden.

Die richtige Zeile der Tabelle wird über die Interrupt-Nummer in der lokalen Kachel (Quell-IRL) aus dem Interrupt-Vektor *IRQ_Out* für den entsprechenden virtuellen Prozessor ausgewählt. Interrupts lassen sich auch an mehrere virtuelle Prozessoren senden (engl. *Multicast*). In dem Fall werden mehrere Interrupt-Pakete nacheinander durch die Netzwerkschnittstelle versendet.

Um eine auf Interrupts basierende Kommunikation über Kachelgrenzen hinweg aufzusetzen, müssen zunächst die Interrupt-Steuerungen in der Quell- und der Zielkachel sowie die Interrupt-Übersetzung in der Quellkachel programmiert werden:

- Zuerst programmiert die Systemsteuerung bei Systemstart oder nach einer Änderung der Zuordnung von Anwendungen und Kacheln die Interrupt-Übersetzungstabelle in der Quellka-

chel. Hierbei werden die Einträge für alle Interrupt-Nummern, die an entfernte Prozessoren weitergeleitet werden sollen, und für jeden virtuellen Prozessor, der diese Interrupts erhalten soll, programmiert.

- Danach werden Quell- und Zielkachel von der Systemsteuerung aktiviert und die entsprechenden Interrupt-Nummern in der lokalen Interrupt-Steuerung der Quellkachel ebenfalls von der Systemsteuerung demaskiert. Normalerweise würden Interrupts durch den Zielprozessor selbst demaskiert. Da der Prozessor in der Zielkachel jedoch keine Kenntnis von einer weiteren Kachel und der darin enthaltenen Interrupt-Steuerung hat, muss die Programmierung der Interrupt-Steuerung in der Quellkachel, anstelle des virtuellen Prozessors in der Zielkachel, ebenfalls durch die Systemsteuerung erfolgen.

- Abschließend wird die Interrupt-Steuerung in der Zielkachel von der dortigen lokalen Anwendung programmiert. Die Interrupt-Nummern des Prozessors in der Quellkachel, der durch die lokale Netzwerkschnittstelle repräsentiert wird, werden dabei genauso behandelt wie andere lokale Interrupts auch.

Ein ausführliches Beispiel zur Kommunikation über Kachelgrenzen hinweg wird in Abschnitt 4.6 erläutert.

Bei der Interrupt-Weiterleitung ist es wichtig, zu verhindern, dass sich Fehler von einer Kachel ausbreiten und zu Fehlern im Netzwerk oder in einer anderen Kacheln führen. Daher werden alle Interrupt-Anfragen, welche die Kachel verlassen, durch die, in Abbildung 4.5 angedeutete und in Abschnitt 4.2.10 genauer erläuterte, Laufzeitüberwachung kontrolliert.

Da der hier vorgestellte Interrupt-Übersetzungsmechanismus unabhängig von lokalen Busprotokollen ist und lediglich darauf basiert, lokale Interrupt-Signale an andere Kacheln weiterzuleiten, können ebenso Kacheln mit anderen Prozessoren und anderen lokalen Bussen eingesetzt werden, um so heterogene Vielkernplattformen zu generieren.

4.2.10. Laufzeitüberwachung

Einer der Kernpunkte dieser Arbeit ist die Laufzeitüberwachung für die IDAMC-Vielkernplattform, die aber in angepasster Form auch für andere Vielkernplattformen nutzbar ist, sofern diese Hardware-Erweiterungen beziehungsweise Hardware-Anpassungen erlauben. Einer der Vorteile einer Vielkernplattform ist, dass sich vorhandene Ressourcen effizient von mehreren Anwendungen gemeinsam nutzen lassen. Hierbei können sich Anwendungen, welche dieselben Ressourcen wie Energieversorgung, Speicherplatz oder Kommunikationskanäle gemeinsam verwenden, jedoch gegenseitig negativ beeinflussen. Welchen Einfluss weniger kritische oder auch nicht sicherheitsrelevante Anwendungen auf die Analyse einer kritischen Anwendung haben und wie dieser Einfluss begrenzt werden kann, wird in Kapitel 3 erläutert.

Die Laufzeitüberwachung, welche in diesem Abschnitt beschrieben wird, dient dazu, Garantien, welche zur Entwurfszeit vor allem für kritische Anwendungen gegeben wurden, zur Laufzeit zu überprüfen und gegebenenfalls zu erzwingen, um so die Anforderungen an weniger kritische Anwendungen reduzieren zu können. Dafür wird die Nutzung von gemeinsamen Ressourcen innerhalb einer Kachel und von den jeweiligen Kacheln ausgehend durch spezielle Aktivitätszähler registriert und im Falle einer Abweichung eine geeignete Reaktionen automatisch initiiert. Hierdurch wird

sichergestellt, dass Garantien, welche einer Anwendung zugesichert wurden, nicht durch andere Anwendungen, welche sich nicht wie spezifiziert verhalten, verletzt werden.

Die Laufzeitüberwachung ist in den Netzwerkschnittstellen, welche die jeweiligen Kacheln und den Rest des Systems miteinander verbinden, implementiert. Dies erfordert, dass Anwendungen unterschiedlicher Kritikalität, welche durch die hier vorgestellte Laufzeitüberwachung separiert werden sollen, zur Fehlereingrenzung auf Prozessoren in unterschiedlichen Kacheln ausgeführt werden müssen. Da sämtliche Laufzeitüberwachungsmechanismen in Hardware implementiert sind, laufen sie parallel zu den eigentlichen Anwendungen und beeinflussen diese daher, außer im Fall einer erkannten Abweichung, nicht. Für weitergehende softwarebasierte Analysen können die gesammelten Daten zusätzlich zu einer automatischen Reaktion durch die Systemsteuerung ausgelesen werden.

Die Laufzeitüberwachung der IDAMC-Plattform wurde vor allem für die Separierung unterschiedlich kritischer Anwendungen entworfen, kann aber auch sowohl zur optimalen Ausnutzung der Plattform, zur Fehlersuche in laufenden Anwendungen als auch zur Charakterisierung von einzelnen Anwendungen verwendet werden, um so beispielsweise die nötigen Daten für die Analyse, welche in Kapitel 3 erläutert wird, zu erhalten.

Gemeinsam verwendete Komponenten

Die Überwachung der Verwendung von gemeinsamen Komponenten, wie zum Beispiel einem externen Speicher oder dem NoC, basiert auf zwei Tabellen, welche eng verbunden sind mit der Adressübersetzung, welche ebenfalls auf zwei Tabellen basiert und in Abschnitt 4.2.8 erläutert wird. Jede Zeile in einer der beiden Übersetzungstabellen für Speicher- und I/O-Adressbereiche entspricht einer Zeile in einer der Tabellen zur Überwachung gemeinsam verwendeter Komponenten. Es ist also jeweils eine Tabelle für die Überwachung kleinerer I/O-Adressbereiche und eine für größere Speicheradressbereiche vorhanden. Eine Zeile in den Tabellen ist damit jeweils einem Adressblock von entweder einem Megabyte für Speicheradressbereiche oder 256 Bytes für I/O-Adressbereiche zugeordnet. Jede Zeile enthält dabei, wie in Abbildung 4.6 dargestellt, die Anzahl an Zugriffen auf den entsprechenden Adressblock im aktuellen Zeitfenster sowie die maximal erlaubte Anzahl an Zugriffen. Die Zeilen sind jeweils 40 Bits breit, wovon jeweils 20 Bits auf die Zähler und 20 Bits auf den Wert für die Maximalanzahl entfallen. Hiermit können also bis zu $2^{20} - 1 = 1048575$ Zugriffe pro Adressblock in jedem Zeitintervall registriert werden. Jeder zusätzlich zu überwachende Adressblock erhöht die Speicheranforderungen der Laufzeitüberwachung damit um lediglich 40 Bits, was den hier vorgestellten Überwachungsmechanismus zu einer skalierbaren Lösung macht, welcher auch für sehr große System eingesetzt werden kann.

Jeder Zugriff auf eine gemeinsam verwendete Komponente erhöht den aktuellen Zählerstand um die Größe des Zugriffs in Anzahl an gelesenen oder geschriebenen Wörtern, welche jeweils 32 Bits umfassen. Die Überwachung der Nutzung einer Komponente, ausgehend von den einzelnen Kacheln, kann entweder periodisch durch die Systemsteuerung oder automatisch in Hardware erfolgen. Im ersten Fall müsste die Systemsteuerung sämtliche zu überwachende Werte periodisch auslesen, zurücksetzen, mit den erwarteten Werten vergleichen und im Falle einer Abweichung, eine entsprechende Reaktion auslösen. Die Reaktionszeit einer solchen Lösung wäre jedoch zum einen schwierig zu bestimmen, wäre möglicherweise sehr groß und würde außerdem mit der Größe des Systems wachsen. Eine zentrale Einheit zur Überwachung ist für Vielkernplattformen daher eher

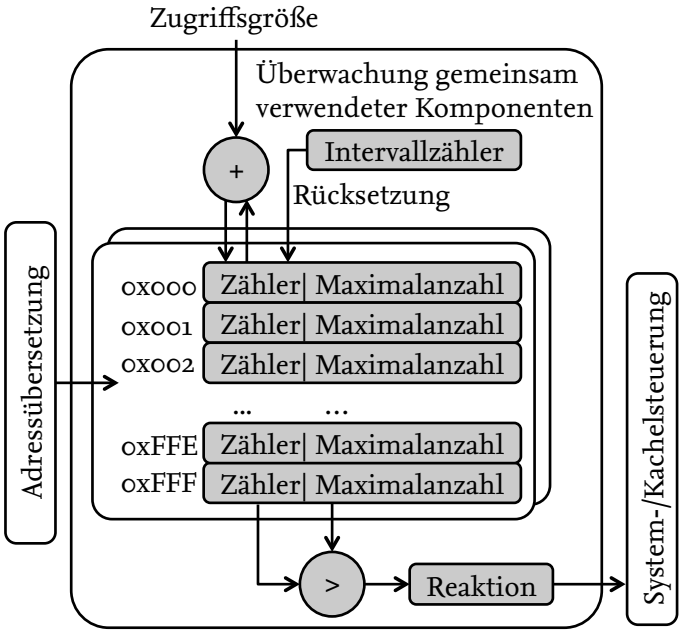

Abbildung 4.6.: Überwachung gemeinsam verwendeter Komponenten

ungeeignet und sollte zumindest für größere Systeme durch eine dezentralisierte und automatische Überwachung ergänzt oder ersetzt werden. Mehr Details zur Reaktionszeit werden in Abschnitt 3.4 erläutert.

Für die hier vorgestellte dezentralisierte Variante in Hardware müssen die in Abbildung 4.6 dargestellten Maximalzählerstände pro Adressregion sowie der Intervallzähler zur Messung des zugehörigen Zeitintervalls zunächst programmiert werden. Die Daten hierfür werden von der Analyse übernommen. Da der Intervallzähler in jedem Takt dekrementiert wird, hängt das größtmögliche Zeitintervall von der Eingangsfrequenz des Überwachungsmoduls ab. Bei einer 100 MHz Eingangsfrequenz könnten daher Intervalle von bis zu 43 Sekunden programmiert werden. Zusätzlich zu dem Intervallzähler und den Maximalzählerständen muss eine geeignete Reaktion ausgewählt werden, für den Fall, dass die maximale Anzahl an Zugriffen auf einen oder mehrere der überwachten Adressbereiche im aktuellen Zeitintervall überschritten wird. Mögliche Reaktion hierfür sind:

- Senden einer Nachricht an die Systemsteuerung, um Fehler zu registrieren oder weitergehende Reaktionen auszulösen, wie beispielsweise die Verschiebung von Anwendungen auf andere Kacheln oder die Abschaltung des Gesamtsystems, um so beispielsweise in einen sicheren Zustand zu gelangen.

- Deaktivieren der Kachel, falls ein permanenter Fehler in der Kachel angenommen wird. Die Deaktivierung einer Kachel ist über ein dauerhaftes Rücksetzen realisiert.

- Neustarten der Kachel, falls ein vorübergehender Fehler angenommen wird und das Neustarten der Kachel wieder zu einem fehlerfreien Zustand führen könnte. Die Kachel wird zunächst nur deaktiviert. Der Neustart darf erst am Ende des aktuellen Zeitintervalls erfolgen, da die erlaubte Höchstanzahl an Zugriffen auf mindestens eine der gemeinsam verwendeten Komponenten im aktuellen Zeitintervall bereits erreicht ist. Der hier beschriebene Neustart einer Kachel

ähnelt einer externen *Watchdog*-Schaltung von herkömmlichen busbasierten Einzel- und Mehrprozessorsystemen, welche die Schaltung aus einem unvorhergesehenem Zustand wieder zurück in einen definierten Anfangszustand bringt. Die vorhandene *Watchdog*-Funktionalität des GPTIMER-Moduls aus der Gaisler-Bibliothek, welche in [4] beschrieben ist, wird hierdurch nicht eingeschränkt und kann zusätzlich verwendet werden.

- Verzögerung der Ausführung bis zum Ende des aktuellen Zeitintervalls, für welches die maximale Anzahl an Zugriffen auf mindestens eine der überwachten Komponenten bereits erreicht ist, um so weitere Zugriffe im aktuellen Zeitintervall zu unterbinden. Zu Beginn des nächsten Zeitintervalls kann die Anwendung da fortgesetzt werden, wo sie unterbrochen wurde.

Die letzten drei der genannten Reaktionen verhindern direkt weitere Zugriffe auf gemeinsam genutzte Komponenten und damit weitere Beeinflussungen von anderen möglicherweise höher kritischen Anwendungen. Das Senden einer Nachricht an die Systemsteuerung kann zusätzlich zu einer automatischen Reaktion aktiviert werden, um weitergehende Aktionen auszulösen oder auch nur zur Registrierung von Abweichungen für eine spätere Analyse. Der Neustart beziehungsweise die Deaktivierung einer Kachel kann auch explizit durch die Systemsteuerung erfolgen, beispielsweise nach Erhalt einer automatischen Nachricht.

Wenn der Intervallzähler abläuft, ohne dass einer der überwachten Werte seinen Maximalzählerstand überschreitet, werden sämtliche Zähler vor Beginn des nächsten Intervalls automatisch zurückgesetzt und der Intervallzähler startet wieder von Neuem.

Die Überwachung der gemeinsam verwendeten Komponenten erhöht die Verzögerung für Zugriffe auf Komponenten außerhalb der jeweiligen Kacheln nicht, da die Aktualisierung der Tabelle sowie der Vergleich der Zählerstände mit den Maximalwerten parallel mit der Adressübersetzung geschieht.

Da die Trennung von unterschiedlich kritischen Anwendungen der eigentliche Grund für den Einsatz der Laufzeitüberwachung von Zugriffe auf gemeinsam verwendete Komponenten ist, werden Auswirkungen auf überwachte weniger oder nicht kritische Anwendungen nicht weiter untersucht. Einschränkungen oder die komplette Deaktivierung von nicht sicherheitsrelevanten Anwendungen, welche sich nicht wie spezifiziert verhalten und so eventuell höher kritische Anwendungen negativ beeinflussen können, werden mit dem hier vorgestellten Mechanismus in Kauf genommen.

Interrupt-Aufkommen

Da Ereignisse, wie eingehende Interrupt-Anfragen, Aufgaben auf dem Zielprozessor aktivieren, wodurch das Zeitverhalten aller anderen dort laufenden Aufgaben beeinflusst wird, muss die Anzahl eingehender Interrupts der Anzahl entsprechen, welche bei der Analyse angenommen wurde. Wie in Kapitel 3 erläutert, wird das erwartete Verhalten durch Ereignismodelle beschrieben, welche unter anderem auch den minimalen Abstand zwischen aufeinanderfolgenden Ereignissen definieren.

Dieser Abstand wird für die hier beschriebene Überwachung des Interrupt-Aufkommens ständig kontrolliert und kann zur Laufzeit erzwungen werden. Dies ist besonders wichtig für Peripheriemodule und Anwendungen, welche Interrupts an Prozessoren senden, die sicherheitskritische Anwendungen ausführen.

Der in Abbildung 4.7 dargestellte Mechanismus basiert ebenfalls, wie die Überwachung von gemeinsam verwendeten Komponenten, auf einer Tabelle, welche hier wiederum eng mit der Interrupt-Übersetzung gekoppelt ist. Die Tabelle enthält einen Satz von jeweils 15 Einträgen für jeden virtuellen

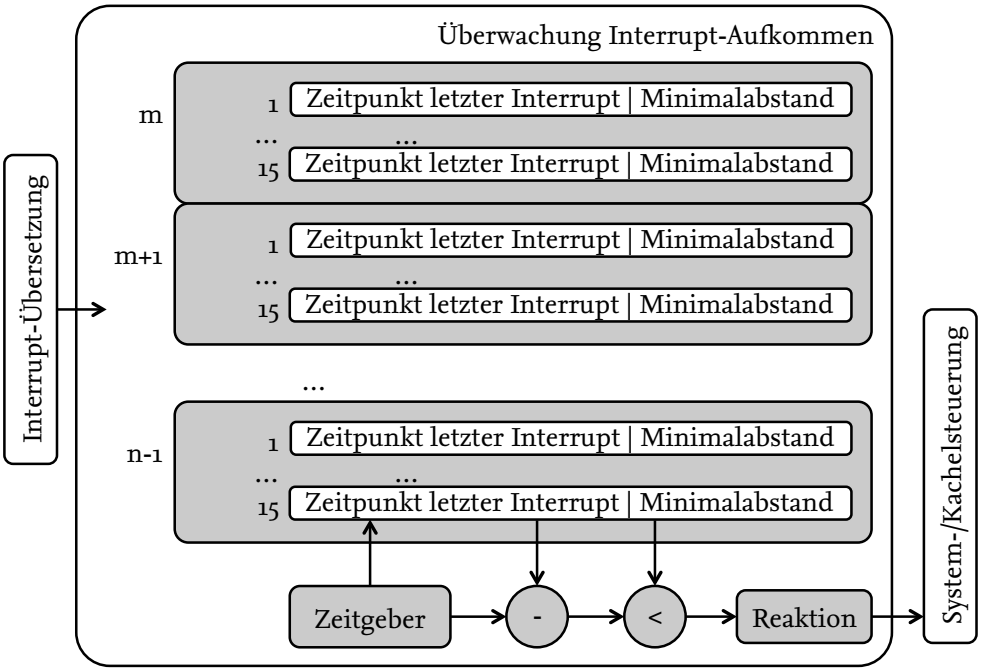

Abbildung 4.7.: Überwachung Interrupt-Aufkommen

Prozessor. Jeder Tabelleneintrag enthält den Zeitpunkt des letzten zugehörigen Interrupts und den geforderten minimalen Abstand zwischen zwei aufeinanderfolgenden Interrupts. Um den Zeitpunkt des letzten Interrupts und den Abstand zwischen aufeinanderfolgenden Interrupts bestimmen zu können, wird ein Zeitgeber benötigt. Für den Fall, dass der minimale Abstand unterschritten wird, kann auch hier eine automatische Reaktion festgelegt werden. Der Mechanismus wird bei Systemstart von der Systemsteuerung konfiguriert und kann auch zur Laufzeit ausschließlich von der Systemsteuerung geändert werden.

Sobald die Überwachung des Interrupt-Aufkommens eine Abweichung des erwarteten Verhaltens erkennt, also sobald der minimale Abstand zwischen zwei aufeinanderfolgenden Interrupt-Anfragen unterschritten wird, löst die ausgewählte Reaktion automatisch aus. Die möglichen Reaktionen sind an den Reaktionen für die Überwachung von gemeinsam verwendeten Komponenten angelehnt, welche in Abschnitt 4.2.10 beschrieben sind. Auch hier kann die betroffene Kachel deaktiviert werden, wenn ein permanenter Defekt angenommen wird, oder bei einem transienten Fehler neu gestartet werden. Ebenso kann eine Nachricht an die Systemsteuerung gesendet werden. Zusätzlich kann der aktuelle Interrupt verzögert werden, um dem bei der Analyse angenommenen Ereignismodell zu entsprechen. Diese Reaktion verhält sich anders als die Verzögerung bei der Überwachung gemeinsam verwendeter Komponenten. Dort wird der aktuell ausgehende Transfer bis zum Ende des aktuellen Zeitintervalls verzögert, was den Prozessor, welcher die betroffene Anfrage gestellt hat, ebenso lange blockiert, da dieser dann so lange auf die Antwort warten muss. Entspricht der Interrupt mit der aktuell höchsten Priorität nicht dem erwarteten Ereignismodell und soll verzögert werden, wird der Interrupt mit der nächst niedrigeren Priorität von der Interrupt-Übersetzung vorgezogen bis der geforderte Mindestabstand für den verzögerten Interrupt erreicht ist.

Das Ziel der programmierbaren Reaktionen ist es, mit Ausnahme der Nachricht an die Systemsteuerung, kommunizierende unterschiedlich kritische Anwendungen so voneinander zu trennen, dass Fehler in weniger kritischen Anwendungen nicht zu Fehlern in höher kritischen Anwendungen führen können. Die Beeinflussung von weniger kritischen Anwendungen auf den überwachten Kacheln wird daher, wie bei der Überwachung von gemeinsam verwendeten Komponenten, in Kauf genommen und nicht weiter untersucht.

Die Zeitpunkte der letzten ausgehenden Interrupts von den einzelnen Kacheln sind auch für die Systemsteuerung zugreifbar, ebenso die manuelle Einleitung der Reaktionen. Es ist also grundsätzlich möglich, die Überwachung des Interrupt-Aufkommens zentral durch die Systemsteuerung durchzuführen, wovon aber aus denselben Gründen, wie in Abschnitt 4.2.10 und 3.4 beschrieben, abgeraten wird. Die hier implementierte dezentralisierte Überwachung in Hardware in den Netzwerkschnittstellen erlaubt eine sehr kurze Reaktionszeit auf ein eventuell abweichendes Verhalten von den Anwendungen auf den einzelnen Kacheln. Die Überwachung des Interrupt-Aufkommens schützt Interrupt-Ziele in anderen Kacheln und verhindert darüber hinaus zusätzliche Belastungen des NoCs, indem jede einzelne zusätzliche Interrupt-Nachricht, welche nicht dem erwarteten Minimalabstand entspricht, bereits an der Quelle unterbunden werden kann.

Da die Überwachung des Interrupt-Aufkommens parallel zu der Interrupt-Übersetzung geschieht, wird durch den Überwachungsmechanismus, wie bei der Überwachung der gemeinsam verwendeten Komponenten in Abschnitt 4.2.10 auch, keine zusätzliche Verzögerung in die Interrupt-Behandlung eingefügt.

Da die Tabelle in Abbildung 4.7 lediglich die Zeitpunkte der jeweils letzten zugehörigen Interrupt-Anfragen speichert, lassen sich nur die Abstände zwischen $k = 2$ aufeinanderfolgenden Ereignissen überwachen und somit nur Minimalabstände periodischer Interrupts erzwingen. Für nicht periodische oder stoßweise (engl. *bursty*) Interrupts oder Interrupts mit *Jitter* ist die Lösung nicht optimal, kann aber, wie in [64] gezeigt, durch die Speicherung weiterer, weiter zurückliegender Interrupts erweitert werden, was jedoch auf Kosten der Speicheranforderungen erreicht würde.

Energieverbrauch

Aktivität in Prozessoren, Speichern, Peripheriemodulen und dem NoC benötigt Energie. Abschätzen lässt sich diese Energie, wie in Abschnitt 3.3.1 erläutert, durch das Zählen von Ereignissen und deren Gewichtung mit der damit verbundenen Energie. In Bezug auf die Trennung unterschiedlich kritischer Anwendungen sind vor allem zwei Gruppen von Ereignissen, welche durch die jeweiligen Anwendungen ausgelöst werden, interessant.

Zum einen muss sich der Gesamtenergieverbrauch pro Anwendung unter einer vorher festgelegten Grenze bewegen, um die verfügbare Energie für andere Anwendungen nicht mehr als angenommen einzuschränken. Hierfür müssen sämtliche Ereignisse, ausgehend von den jeweiligen Anwendungen, registriert und gewichtet werden, unabhängig davon, wo auf dem System die Energie verbraucht wird, also lokal in derselben Kachel wie der ausführende Prozessor, im NoC oder auch in Peripheriemodulen und Speichern in anderen Kacheln. Zum anderen darf die Energiedichte, und damit die Temperatur, in keiner Region des Chips eine bestimmte Grenze überschreiten, da ansonsten benachbarte Komponenten beeinflusst werden können oder gegebenenfalls die Zuverlässigkeit oder Lebensdauer des gesamten Chips reduziert wird [79]. Eine hohe Energiedichte in einer Kachel kann durch eine lokal ausgeführte Anwendung hervorgerufen werden, oder aber auch durch eine

Anwendung auf einem Prozessor in einer anderen Kachel, welche übermäßig häufig auf eine Speicherschnittstelle oder ein Peripheriemodul in der lokalen Kachel zugreift. Um eine erhöhte Energiedichte zu erkennen, müssen daher alle Ereignisse innerhalb der einzelnen Kacheln überwacht werden, unabhängig davon, auf welchem Prozessor die verursachenden Anwendungen implementiert sind, also auf einem lokalen Prozessor oder einem Prozessor in einer anderen Kachel.

Durch die Zuordnung von Ereignissen zu einzelnen Anwendungen und deren Gewichtung lässt sich, wie bereits in Kapitel 3 beschrieben, das Gesamtenergiebudget sowie die erlaubte Energiedichte in den jeweiligen Chip-Regionen, wie andere gemeinsam verwendete Ressourcen auch, auf die einzelnen Anwendungen aufteilen, überwachen und gegebenenfalls auch erzwingen. So kann ein erhöhter Energieverbrauch einer weniger kritischen Anwendung erkannt und begrenzt werden, bevor eine höher kritische Anwendung negativ beeinflusst wird, um so die Trennung von unterschiedlich kritischen Anwendung auch in Bezug auf den Energieverbrauch sicherzustellen.

Die hier beschriebene Überwachung des Energieverbrauchs ist zusammen mit der Überwachung von gemeinsam verwendeten Komponenten und des Interrupt-Aufkommens in den Netzwerkschnittstellen implementiert, um die Energiedichte innerhalb der Kacheln und den von einer Kachel ausgehenden Energieverbrauch abzuschätzen. Fehler können so auf einzelne Kacheln begrenzt werden, weshalb es aber auch erforderlich ist, Anwendungen unterschiedlicher Kritikalität auf Prozessoren in unterschiedlichen Kacheln zu implementieren.

Die Überwachung des Energieverbrauchs basiert, wie in Abbildung 4.8 dargestellt, hauptsächlich auf vier Tabellen, welche Energiegewichtungen für bestimmte Ereignisse enthalten. Zwei der Tabellen sind eng mit der Adressübersetzung gekoppelt und beinhalten Energiegewichtungen für Zugriffe auf Speicheradressbereiche beziehungsweise I/O-Adressbereiche in anderen Kacheln. Zwei weitere Tabelle enthalten Energiegewichtungen einmal für lokale Prozessoren und andere Bus-Master und einmal für lokale Speicher und Peripheriemodule. Die entsprechenden Tabelleneinträge werden bei den beiden zuerst genannten Tabellen über Zugriffe auf die Adressübersetzung und bei den zuletzt genannten Tabellen hauptsächlich über Signale der lokalen AHB- und APB-Busse aktiviert. Um bei der Implementierung FPGA-Ressourcen zu schonen, sind die beiden Tabellen für Komponenten außerhalb der lokalen Kachel gemeinsam in einem BRAM implementiert und die Tabelle für lokale Master ist mit Registern realisiert.

Die Energiegewichtungen in den Tabellen enthalten jeweils die gesamte benötigte Energie für einen Zugriff auf eine Komponente, inklusive der Energie, die auf lokalen Bussen, in Netzwerkschnittstellen und dem NoC verbraucht wird. Daher können sich die Energiegewichtungen von identischen Komponenten dennoch unterschieden, wenn sie von der aktivierenden Kachel unterschiedlich weit entfernt sind.

Um die Genauigkeit der Energieabschätzung vor allem für lokale Prozessoren zu erhöhen, könnten einzelnen Prozessoren auch mehrere Ereignisse zugeordnet werden, beispielsweise für Aktivität in der Fließkommaeinheit oder in einem Co-Prozessor. Die Experimente, welche in Kapitel 5 präsentiert werden, zeigen jedoch, dass die Unterschiede des Energieverbrauchs für unterschiedliche Operationen innerhalb des LEON3-Prozessors, beispielsweise gegenüber Zugriffen auf Speicherschnittstellen in anderen Kacheln, vernachlässigbar sind. Daher enthält die Tabelle mit den Energiegewichtungen für lokale Master lediglich einen einzelnen Eintrag pro Master. Wenn für unterschiedliche Ereignisse in einer Komponente eine gemeinsame Energiegewichtung verwendet wird, ist es jedoch wichtig, hier den größten Wert zu verwenden, damit der Wert der Energieabschätzung mindestens so hoch ist

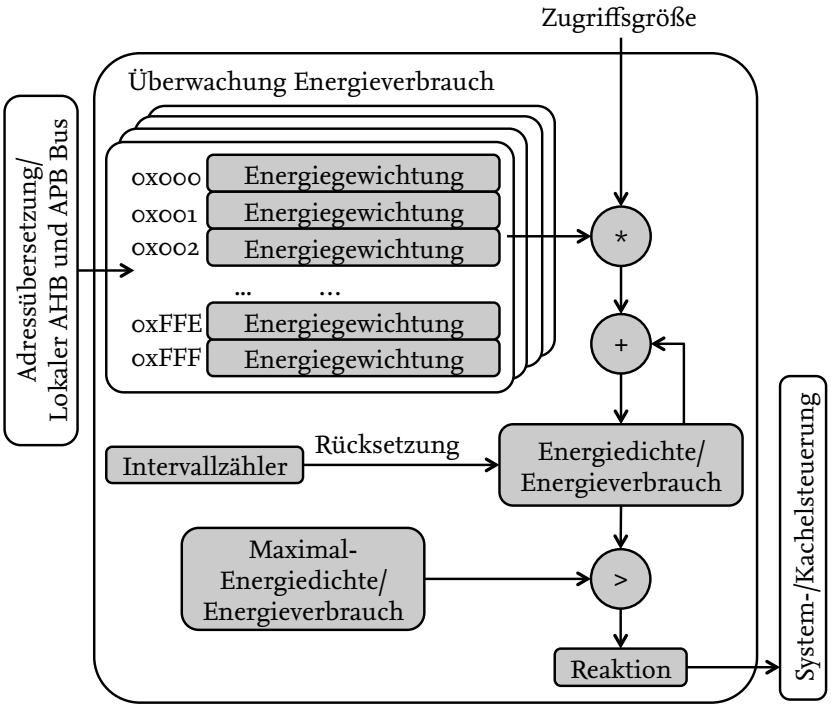

Abbildung 4.8.: Überwachung Energieverbrauch

wie die tatsächlich verbrauchte Energie. Nur so kann eine sichere Trennung des Energieverbrauchs von unterschiedlich kritischen Anwendungen erreicht werden.

Bei der Detektierung eines Ereignisses wird die zugehörige Menge an Energie mit der Größe des Zugriffs in Anzahl an 32-Bit Wörtern multipliziert und aufsummiert. Die Werte werden in zwei unterschiedlichen Registern aufsummiert, einmal für Energie, die in der lokalen Kachel verbraucht wird, und einmal für Energie, die von einer Anwendung auf einem lokalen Prozessor benötigt wird, unabhängig davon, ob in einer lokalen Komponente oder außerhalb in einer anderen Kachel oder dem NoC.

Für die Bestimmung der Energiedichte innerhalb einer Kachel werden alle Ereignisse in lokalen Mastern und in lokalen Slaves aufsummiert. Von welcher Anwendung diese Aktivität initiiert wurde, ist hierbei unwichtig. Für die Abschätzung des Energieverbrauchs, welcher von einer lokalen Kachel ausgeht, werden dagegen alle Ereignisse gewichtet und aufsummiert solange sie von einem lokalen Master ausgehen. Hierbei ist es unwichtig, wo die Energie verbraucht wird, also in einem lokalem Master oder Slave oder in einer entfernten Komponente, die zu einem der Speicheradressbereiche oder I/O-Adressbereiche gehört, welche durch die Adressübersetzung lokal eingeblendet werden.

Um auf der IDAMC-Plattform unterschiedlich kritische Anwendungen nicht nur zeitlich und räumlich, sondern auch deren Energieverbrauch sicher voneinander trennen zu können, müssen die Energiegewichtungen, vor allem für Kacheln, welche weniger kritische Anwendungen ausführen, für alle von einer lokalen Anwendung erreichbare Komponenten programmiert werden. Weiterhin muss die maximal erlaubte Energiedichte innerhalb einer Kachel, der maximale Gesamtenergieverbrauch, der von einer Kachel ausgehen darf, das zugehörige Zeitintervall sowie eine von vier möglichen automatischen Reaktionen von der Systemsteuerung festgelegt werden. Die Energiegewichtungen erhält

man aus der Charakterisierung der Plattform, welche in Abschnitt 3.3 erläutert und in Abschnitt 5.2.1 experimentell durchgeführt wird. Die maximale Energiedichte, der maximale Energieverbrauch und das zugehörige Zeitintervall wird durch die Analyse des Energieverbrauchs aller Anwendungen der Plattform, welche in Abschnitt 3.3 beschrieben wird, bestimmt.

Wenn der Intervallzähler abläuft, ohne dass eines der beiden Energiebudgets ausgeschöpft wird, werden die beiden Akkumulatoren automatisch zurückgesetzt. Wenn jedoch eines der Budgets innerhalb des gegebenen Zeitintervalls durch eine erhöhte Aktivität überreizt wird, beispielsweise durch eine fehlerhafte Anwendung oder unerwartete Eingangsdaten, kann eine geeignete Reaktion automatisch ausgelöst werden. Die möglichen Reaktionen entsprechen weitestgehend den Reaktionen, welche auch für die Überwachung von gemeinsam verwendeten Komponenten und des Interrupt-Aufkommens zur Verfügung stehen und weiter oben genauer erläutert werden. Es kann eine Nachricht an die Systemsteuerung gesendet werden, die betroffene Kachel kann zurückgesetzt werden oder auch komplett deaktiviert werden. Weiterhin kann die betroffene Kachel angehalten werden, um einen weiteren dynamischen Energieverbrauch bis zum Ende des aktuellen Zeitintervalls zu verhindern. Am Ende des Zeitintervalls wird die Anwendung dort fortgeführt wo sie unterbrochen wurde. Für diese Reaktion würde sich in bestimmten Fälle auch anbieten, den Takt oder die Energieversorgung komplett abzuschalten, um weitere Energie einsparen zu können (engl. *Clock Gating* und *Power Gating*). Da zum einen das Ziel des hier vorgestellten Überwachungsmechanismus nicht die Optimierung des Energieverbrauchs ist, sondern die Trennung von unterschiedlich kritischen Anwendungen, und zum anderen die Abschaltung des Taktes und der Versorgungsspannung von einzelnen Chip-Regionen sich mit zum Zeitpunkt dieser Arbeit verfügbaren FPGAs entweder gar nicht oder nur sehr eingeschränkt umsetzen lässt, wurden diese und weitere Energiesparmaßnahmen wie *Voltage Scaling* und *Frequency Scaling* zur Variation der Spannung und der Frequenz nicht weiter betrachtet.

Nach Empfang einer Nachricht können die Reaktionen Neustarten, Deaktivieren und Anhalten, um weiteren dynamischen Energieverbrauch innerhalb der betroffenen Kachel oder von dieser ausgehend zu unterbinden, auch von der Systemsteuerung explizit ausgeführt werden. Durch die Netzwerkverzögerung und Ausführungszeit der Steuerungsanwendung würde die Reaktionszeit, um weiteres Überschreiten des festgesetzten Budgets zu verhindern, deutlich gegenüber einer dezentralisierten Hardware-Lösung ansteigen und auch mit größeren Systemen wachsen. Eine zentrale Lösung ist für die Separierung des Energieverbrauchs von unterschiedlich kritischen Anwendungen daher genauso wenig geeignet wie für die Überwachung gemeinsam verwendeter Komponenten oder des Interrupt-Aufkommens. Die hier vorgestellte Implementierung in Hardware in den Netzwerkschnittstellen kann bereits das erste Ereignis, welches zur Überschreitung eines der Energiebudgets führen würde, unterbinden. Diese Reaktionszeit kann garantiert werden und wächst auch selbst für sehr große Systeme nicht.

Da der Mechanismus zur Energieüberwachung parallel zu der Adressübersetzung beziehungsweise Aktivitäten in lokalen Komponenten implementiert ist, fügt er, wie die Laufzeitüberwachung von gemeinsam verwendeten Komponenten und des Interrupt-Aufkommens, keinerlei zusätzliche Verzögerung in den Ablauf der überwachten Anwendungen ein. Die Beeinflussung von weniger kritischen oder nicht sicherheitsrelevanten Anwendung durch eine automatische Reaktion bei einer erkannten Abweichung des spezifizierten Verhaltens wird jedoch auch hier für die sichere Separierung von höher kritischen Anwendungen akzeptiert.

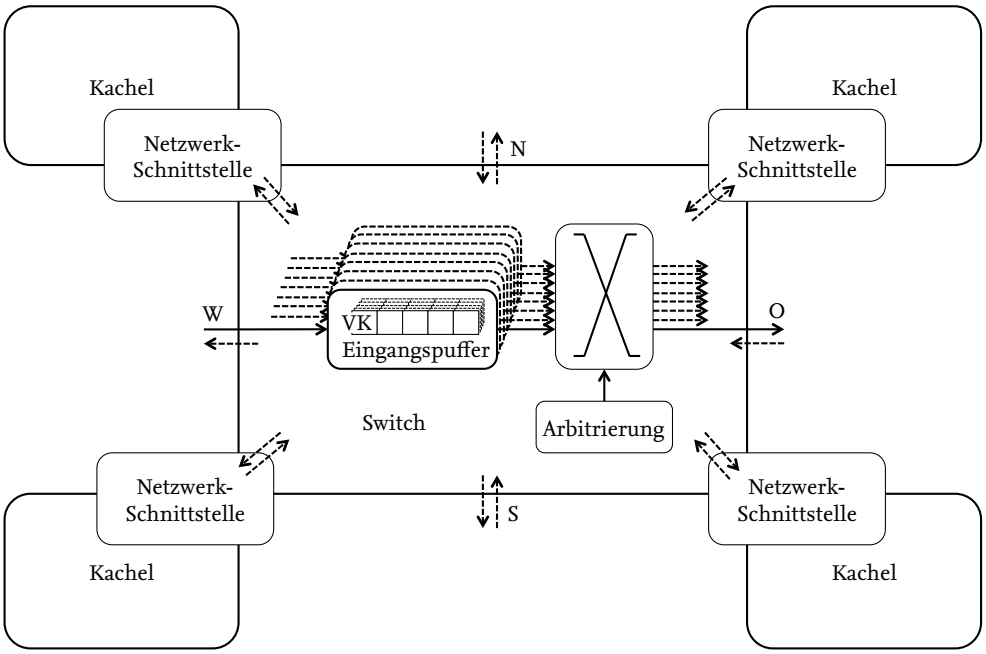

Abbildung 4.9.: Switch

4.3. Network-on-Chip

Mehrere auf der GRLIB-Bibliothek von Aeroflex Gaisler basierende Teilsysteme, genannt Kacheln, wurden im Rahmen dieser Arbeit unter Verwendung einer angepassten Version[1] des NoCs aus [56] zu einer Vielkernplattform verbunden. Das IDAMC NoC besteht aus bis zu 64 konfigurierbaren Switches, die über bis zu acht Vollduplex-Verbindungen von jeweils 35 Bits miteinander, oder über Netzwerkschnittstellen mit angeschlossenen Kacheln, verbunden sind. Sowohl die Anzahl der Verbindungen als auch die Netzwerktopologie lassen sich vor der Implementierung frei festlegen. Da sich die einzelnen Ports der Switches untereinander nicht unterscheiden, sich also sowohl für Verbindungen mit Nachbar-Switches als auch zum Anschließen von Kacheln verwenden lassen, können sämtliche Topologien erzeugt werden, welche mit Switches mit bis zu acht Verbindungen gebildet werden können, zum Beispiel ein vermaschtes Netz (engl. *Mesh*), ein Ring, ein Stern oder auch eine Kombination aus diesen. In einem zweidimensionalen vermaschten Netz, wie in der Plattformübersicht in Abbildung 4.1 beispielhaft dargestellt, werden vier Verbindungen verwendet, um benachbarte Switches in Nord-,Ost-, Süd- und Westrichtung anzuschließen. Mit den verbliebenen bis zu vier Verbindungen können dann, wie in Abbildung 4.9 gezeigt, bis zu vier Kacheln angeschlossen werden.

Auf der IDAMC-Plattform wird tabellenbasiertes Routing eingesetzt, dies bedeutet, dass die Route von einer zu einer anderen Kachel in der Quelle festgelegt wird, und zwar in den Adress- beziehungsweise Interrupt-Übersetzungstabellen, welche in Abschnitt 4.2.8 und 4.2.9 beschrieben sind. Tabellenbasiertes Routing erlaubt es der Systemsteuerung, die Paketrouten flexible, aber deterministisch festzulegen und diese auch zur Laufzeit für eventuelle Optimierungen oder Fehlerbehandlungen zu ändern. Pfade können damit so gewählt werden, dass sie sich möglichst wenig überlappen, um

[1] Die Anpassung des NoCs ist durch Jonas Diemer erfolgt

Konkurrenzsituationen in Switches so weit wie möglich zu reduzieren oder auch um fehlerhafte Switches zu umgehen.

Daten werden als Pakete und unter Verwendung des *Wormhole-Switching*-Verfahrens über das NoC übertragen. Bei dem *Wormhole-Switching*-Verfahren werden Pakete in kleinere Teile, welche *Flow Control Digits (Flits)* genannt werden, aufgeteilt. Das erste *Flit* legt die Route für das Paket in den Switches auf dem Weg von der Quelle zur Zielkachel fest und die restlichen *Flits* folgen auf dieser Route. Eingehende *Flits* werden in den Switches in Eingangspuffern zwischengespeichert und von einem Arbiter *Flit* für *Flit* weitergeschaltet.

Wie in Abbildung 4.9 dargestellt, existiert jeweils pro Port des Switches ein Satz an separaten Eingangspuffern, welche verwendet werden, um virtuelle Kanäle (VK) zu realisieren. Virtuelle Kanäle sind separate Puffer, welche jedoch die Verbindungen zwischen den Switches mit den anderen virtuellen Kanälen gemeinsam verwenden. Sowohl die Anzahl an virtuellen Kanälen als auch die Größe der Eingangspuffer kann vor der Implementierung über die zentrale Konfigurationsdatei festgelegt werden. Der verwendete virtuelle Kanal für ein Paket wird ebenfalls wie die Route von der Systemsteuerung in den Adress- beziehungsweise Interrupt-Übersetzungstabellen vorgegeben. Pakete von unterschiedlichen Anwendungen können so voneinander separiert werden, da sie unabhängig voneinander weitergeleitet werden. Selbst wenn ein Paket, welches über einen anderen virtuellen Kanal, jedoch über dieselbe Verbindung gesendet wird, blockiert, können andere Pakete durch die logische Trennung weiter Fortschritte machen.

Um die Bandbreite sowie die Verzögerung innerhalb des NoCs für kritische Anwendungen garantieren zu können, kann diesen ein exklusiver Zugang zu einem der virtuellen Kanäle gewährt werden (engl. *Guaranteed Service*), während sich weniger kritische Anwendungen einen Kanal teilen und so um Ressourcen konkurrieren müssen (engl. *Best Effort*). Die maximale Anzahl an überlappenden Verbindungen mit garantierten Verzögerungen und Bandbreiten ist jedoch durch die Anzahl der vorhandenen virtuellen Kanäle begrenzt.

Die Arbitrierung zwischen unterschiedlichen virtuellen Kanälen verwendet das Rundlauf-Verfahren (engl. *Round Robin Scheduling*). So werden *Flits* in unterschiedlichen virtuellen Kanälen nacheinander zu gleichen Anteilen weitergeleitet. Um Pakete unterschiedlich kritischer Anwendungen mit unterschiedlichen Prioritäten bei der Weiterleitung zu gewichten, um beispielsweise Anwendungen mit einer höheren Kritikalität zu bevorzugen oder den Gesamtdurchsatz zu optimieren, können Verfahren wie *Back Suction* [32] angewendet werden. Bei *Back Suction* werden beispielsweise weniger kritische Verbindungen bevorzugt, solange höher kritische Anwendung gerade noch ihre Anforderungen erfüllen können, um so den Durchsatz von weniger kritischen Verbindungen zu erhöhen.

4.4. Systemsteuerung

Der Großteil der Konfigurationsmöglichkeiten, um Ressourcen der Vielkernplattform an Anwendungen zuzuweisen und die Separierung von unterschiedlich kritischen Anwendungen sicherzustellen, sind in den Netzwerkschnittstellen zusammengefasst. Über die Netzwerkschnittstellen werden entfernte Speicherschnittstellen und Peripheriemodule in anderen Kacheln für lokale Anwendungen erreichbar gemacht, die Kommunikation mit entfernten Prozessoren ermöglicht, vorhersagbare Kommunikationswege durch das NoC definiert sowie sämtliche Überwachungsmaßnahmen, um unterschiedlich kritisch Anwendungen sicher zu trennen, eingeleitet.

Die Konfigurationsmöglichkeiten haben einen Einfluss auf die Stabilität und die Effizienz des Systems, jedoch vor allem auf die Eingrenzung von weniger kritischen Anwendungen, besonders im Fall einer Abweichung. Sie dürfen daher nur für eine Instanz höchster Sicherheitsintegrität zugreifbar sein, also in keinem Fall für eine lokal auf einer Kachel laufende Anwendung, die lediglich die Anforderungen eines geringeren Sicherheitsintegritätslevels erfüllt. Um eine Funktion und deren Überwachung zu trennen, wie in der Norm IEC 61508 [51] empfohlen, sollte darüber hinaus selbst hoch kritischen lokalen Anwendungen kein Zugriff auf die Überwachungsmaßnahmen ihrer zugewiesenen Kachel gewährt werden.

Auf der IDAMC-Plattform ist daher eine der Kacheln als Systemsteuerung festgelegt. Nur von dieser Kachel mit der Identifikationsnummer 0 lassen sich die Konfigurations- und Überwachungsmechanismen unter Berücksichtigung der Anforderungen der jeweiligen Anwendungen programmieren. Konfigurationspakete mit einer anderen Identifikationsnummer werden in den Netzwerkschnittstellen, genauso wie Zugriffe durch lokale Anwendungen, verworfen und nicht an die Adress- oder Interrupt-Übersetzung, die Laufzeitüberwachung oder die Kachelsteuerung weitergeleitet.

Die Systemsteuerung selbst benötigt Zugriff auf die Konfigurationsmöglichkeiten ihrer eigenen Netzwerkschnittstelle, um sich die Konfigurationsadressbereiche der anderen Kacheln sowie beliebige anderer Speicher- und I/O-Adressbereiche in ihren eigenen lokalen Adressbereich einblenden zu können, um so vollen Zugriff auf das System zu erhalten. Die Kachel, welche die Systemsteuerung darstellt, ist daher die einzige Kachel des Systems, bei der einer lokalen Anwendung Zugriff auf die Konfigurationsmöglichkeiten der eigenen Netzwerkschnittstelle gewährt wird.

Die Programmierung der verteilten Konfigurationsmöglichkeiten durch die Systemsteuerung geschieht normalerweise beim Hochfahren des Systems, kann aber auch dynamisch zur Laufzeit erfolgen, um auf unerwartete oder wechselnde Bedingungen zu reagieren. Da die Systemsteuerung damit Einfluss auf sämtliche auf der Plattform laufende Anwendungen hat, auch auf kritische Anwendungen, muss die Systemsteuerung die höchsten Sicherheitsanforderungen, die für irgendeine Anwendung auf der Plattform gelten, ebenfalls erfüllen. Für höhere Sicherheitsintegritätsstufen könnte die Systemsteuerung auch redundant auf mehreren Kacheln implementiert werden. Weiterhin würde es sich für sehr große Systeme anbieten, mit mehreren Kacheln eine mehrstufig Systemsteuerung aufzusetzen, mit jeweils einer Kachel als Teilsystemsteuerung für eine Gruppe von Kacheln.

Auch wenn alle Daten der Laufzeitüberwachung in den Netzwerkschnittstellen der einzelnen Kacheln für die Systemsteuerung ebenfalls lesbar sind, ist es nicht ratsam, die Überwachung selbst von der Systemsteuerung durchführen zu lassen. Für große Systeme wächst die Zahl der zu überwachenden Ressourcen, so dass sehr viele Werte periodisch ausgelesen werden müssten. Durch das Auslesen, das Vergleichen mit den erwarteten Werten sowie die Ausführung einer möglichen Reaktion, würde die Zeit, um, wie in Abschnitt 3.4 erläutert, auf einen Fehler zu reagieren mit der Größe des Systems ansteigen.

Durch die verteilten Überwachungsmechanismen in den Netzwerkschnittstellen, welche ausschließlich durch die Systemsteuerung konfiguriert werden dürfen, wird die zentrale Systemsteuerung von der kontinuierlichen Überwachung der ausgeführten Anwendungen befreit. Sobald die dezentrale Laufzeitüberwachung durch die Systemsteuerung programmiert ist, erfordern deren automatische Reaktionen auf eine Abweichung einer weniger oder nicht sicherheitsrelevanten Anwendung keine

weitere Interaktion mit der Systemsteuerung. Dennoch hat die Systemsteuerung die volle Kontrolle über die gesamte Vielkernplattform und alle darauf ausgeführten Anwendungen.

4.5. Virtualisierung

Die Virtualisierung auf der IDAMC-Plattform unterscheidet sich in einigen Punkten von dem, was üblicherweise unter Virtualisierung verstanden wird. Virtualisierung im herkömmlichen Sinn stellt einer Anwendung exklusiven Zugriff auf alle benötigten Ressourcen zur Verfügung, auch wenn die Anwendung auf der tatsächlich vorhandenen Hardware-Architektur gar nicht lauffähig ist oder die Ressourcen mit weiteren Anwendungen geteilt werden müssen. Weiterhin können durch die herkömmliche Virtualisierung Zugriffe auf kritische Bereiche modifiziert oder auch komplett verhindert werden. Weitere Details zu den klassischen Konzepten der Virtualisierung werden in Abschnitt 2.4 erläutert.

Durch die große Anzahl an Ressourcen einer Vielkernplattform, wie der IDAMC-Plattform, ist es nicht nötig, die Rechenleistung von einzelnen Prozessoren und andere Komponenten auf mehrere virtuelle Maschinen aufzuteilen und diesen dabei einen exklusiven Zugriff vorzutäuschen. Auf einer Vielkernplattform sind im Allgemeinen ausreichend Ressourcen vorhanden, um jeder Anwendung tatsächlich exklusiven Zugriff auf alle benötigten Ressourcen zu geben. Wo dies nicht möglich ist, wie beispielsweise bei externen Speicherschnittstellen, die auch auf einer Vielkernplattform begrenzt sind, können diese Ressourcen, wie bei der herkömmlichen Virtualisierung, transparent zwischen mehreren Anwendungen aufgeteilt werden. Auch für Anwendungen, die nicht für die SPARC-V8-Architektur entwickelt wurden, auf welcher der verwendete LEON3-Prozessor der GRLIB-Bibliothek basiert, und damit nicht direkt auf der IDAMC-Plattform lauffähig sind, können die Virtualisierungsmechanismen der IDAMC-Plattform über herkömmliche Virtualisierunglösungen ergänzt werden.

Die Virtualisierung auf der IDAMC-Plattform bietet mehreren Anwendungen jeweils die Sicht auf ein exklusiv genutztes busbasiertes Einzel- oder Mehrprozessorsystem mit allen benötigen Speicherschnittstellen und Peripheriemodulen, die in Aeroflex Gaislers GRLIB-Bibliothek vorhanden sind oder hinzugefügt werden. Welche Ressourcen einer Anwendung dann tatsächlich zugewiesen werden, wo diese sich auf der Plattform befinden und wie eine Anwendung gegebenenfalls über mehrere Kacheln verteilt wird, bleibt vor den Anwendungen selbst verborgen. Sämtliche Prozessoren, Speicherschnittstellen, Peripheriemodule und die Verbindungen zwischen diesen Komponenten werden auf der IDAMC-Plattform *virtualisiert*. Auf diese Weise lassen sich bestehende Anwendungen ohne Software-Änderungen auf der IDAMC-Plattform integrieren. Auch die Entwicklung von neuen Anwendungen wird hierdurch stark vereinfacht. Weiterhin kann die Systemsteuerung die Zuordnung von Kacheln und verteilten Komponenten zur Laufzeit ändern, um beispielsweise die Verfügbarkeit von einzelnen Anwendungen über das Ersetzen von fehlerhaften Komponenten zu erhöhen oder auch zur Optimierung des Systems.

Die im Rahmen der vorliegenden Arbeit entwickelte Virtualisierungslösung basiert hauptsächlich auf Hardware-Mechanismen in den Netzwerkschnittstellen, welche sich über die Systemsteuerung beim Systemstart oder auch zur Laufzeit programmieren lassen. Hierdurch kann der Rechenaufwand für die Virtualisierung sowie der Anteil des Codes, welcher zu einem Komplettausfall des gesamten Systems führen kann, im Vergleich zu herkömmlichen meist auf Software basierenden Virtualisierunglösungen minimiert werden.

Abhängig von den Anforderungen der einzelnen Anwendungen in Bezug auf verwendete Prozessoren, Speicherschnittstellen und Peripheriemodule, auf benötigte Verbindungsbandbreiten, Verzögerungszeiten und Energie sowie auf zusätzliche Anforderungen wie Fehlertoleranz und Verfügbarkeit weist die Systemsteuerung den Anwendungen entsprechende Ressourcen zu und stellt die ausreichende Separierung der einzelnen Anwendungen sicher. Die Systemsteuerung weist den Anwendungen Kacheln zu und blendet Speicherschnittstellen, Peripheriemodule und weitere Prozessoren über die Adress- und Interrupt-Übersetzung, welche in Abschnitt 4.2.8 und 4.2.9 erläutert werden, in den lokalen Adressraum der Kachel ein und programmiert benötigte Routen und virtuelle Kanäle des NoCs.

Hierbei werden unterschiedlich kritische Anwendungen zur Fehlereingrenzung auf unterschiedlichen Kacheln platziert und deren kachelübergreifende Kommunikation über separate virtuelle Kanäle realisiert. Die sichere Separierung von gemeinsam verwendeten Ressourcen, wie Speicherschnittstellen oder Energie, sowie die sichere Kommunikation zwischen Anwendungen auf unterschiedlichen Kacheln wird über die in Abschnitt 4.2.10 beschriebenen Laufzeitüberwachungsmechanismen sichergestellt.

Um die Zuverlässigkeit oder Verfügbarkeit von einzelnen Anwendungen zu erhöhen, könnten diese auch, ohne dafür vorgesehen zu sein und ohne deren Kenntnis, redundant auf mehreren Kacheln ausgeführt werden, über redundante Routen im NoC kommunizieren oder auch redundante Speicherbereiche verwenden. Der Schwerpunkt der vorliegenden Arbeit liegt jedoch auf der sicheren Separierung von unterschiedlich kritischen Anwendungen. Die Erhöhung der Zuverlässigkeit von einzelnen Anwendungen wurde daher nicht weiter untersucht und bietet Möglichkeiten für weitere Forschungsthemen.

4.6. Kommunikation

Die Kommunikation zwischen Anwendungen auf unterschiedlichen Kachel wird auf der IDAMC-Plattform über gemeinsam verwendete Speicherbereiche realisiert, welche wiederum auch in einer dritten Kachel vorhanden sein können. Die Kommunikation ist durch die in diesem Kapitel bereits beschriebenen Mechanismen flexibel und transparent gestaltet, so dass es für die Anwendungen keinen Unterschied macht, ob die Kommunikation zwischen Anwendungen in unterschiedlichen Kacheln oder in derselben Kachel stattfindet. Ein gemeinsamer Speicherblock wird dazu von der Systemsteuerung über die Adressübersetzung in den jeweiligen Netzwerkschnittstellen für beide Anwendungen in deren lokalen Adressbereich eingeblendet. Die Benachrichtigung über neue Daten in dem gemeinsamen Speicher geschieht über explizite Interrupts, welche zwischen den Anwendungen über die Interrupt-Übersetzung ausgetauscht werden. Sowohl die Benutzung des gemeinsamen Speichers als auch die Anzahl der Interrupt-Anfragen zwischen den Anwendungen können überwacht werden, um so die Kommunikation nicht nur flexibel und transparent, sondern auch sicher zu gestalten.

Ein Anwendungsentwickler hat hierbei stets die Sicht auf ein herkömmliches busbasiertes Multiprozessorsystem, wie in Abbildung 4.2 dargestellt, bei dem die kommunizierenden Prozessoren sowie der gemeinsame Speicher logisch über einen gemeinsamen Bus verbunden sind. Auch wenn die beteiligten Komponenten physisch über mehrere Kacheln des Systems verteilt sind, müssen die beteiligten Anwendungen nicht angepasst werden. Die Aufteilung von miteinander kommunizierenden Anwendungen auf mehrere Kacheln ist vor allem für unterschiedlich kritische Anwendungen

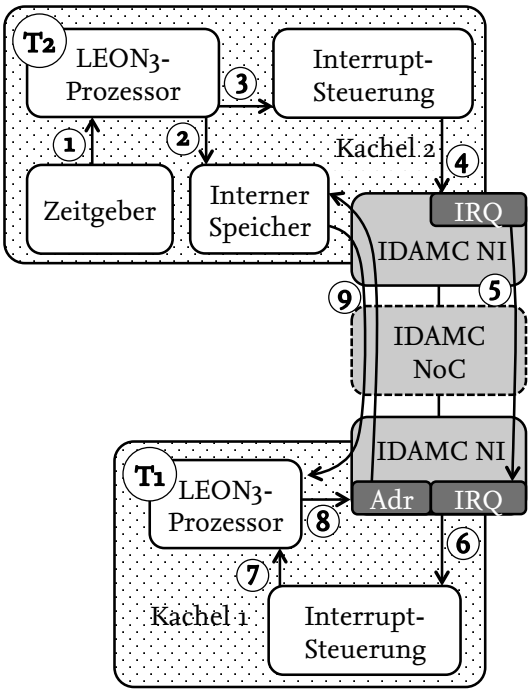

Abbildung 4.10.: Kachelübergreifende Kommunikation (Beispiel)

wichtig, um Fehler in weniger kritischen Anwendungen isolieren zu können. Weiterhin können so Anwendungen zur Laufzeit individuell auf andere Kacheln migriert werden. Indem der gemeinsam verwendete Speicher in einer weiteren Kachel implementiert wird, kann dieser zur Laufzeit transparent ersetzt werden, um so die Verfügbarkeit der realisierten Funktion zu erhöhen.

Abbildung 4.10 zeigt ein Beispiel, wie die kachelübergreifende Kommunikation auf der IDAMC-Plattform mittels Adress- und Interrupt-Übersetzung umgesetzt werden kann. In dem dargestellten Beispiel werden zwei Anwendungen, welche über die Aufgaben T1 und T2 miteinander kommunizieren, von der Systemsteuerung auf Prozessoren in zwei unterschiedlichen Kacheln zugewiesen. Der für die Kommunikation gemeinsam verwendete Speicher befindet sich zusammen mit der Aufgabe T2 in Kachel 2. Damit auch die Aufgabe T1 in Kachel 1 auf diesen Speicher zugreifen kann, wird der entsprechende Adressbereich in Kachel 2 durch die Systemsteuerung über die Programmierung der Adressübersetzung in der Netzwerkschnittstelle der Kachel 1 in deren lokalen Adressbereich eingeblendet. Damit kann Aufgabe T1 auf den Speicher in Kachel 2, oder einen Teil davon, wie auf einen lokalen Speicher zugreifen.

Da Aufgabe T2 in ihrer lokalen Kachel einen zweiten logischen Prozessor als Kommunikationspartner erwartet, der sich physisch jedoch in einer anderen Kachel befindet, muss der Interrupt-Übersetzungsmechanismus in der Netzwerkschnittstelle der Kachel 2 zunächst durch die Systemsteuerung programmiert werden. Die Aufgabe in Kachel 2 kann dann über ihre lokale Interrupt-Steuerung Interrupts für den Prozessor in Kachel 1, welcher in Kachel 2 durch die Netzwerkschnittstelle repräsentiert wird, wie für einen weiteren lokalen Prozessor auslösen.

Da jeder Prozessor die für ihn bestimmten Interrupts in der jeweiligen lokalen Interrupt-Steuerung selbst demaskiert und priorisiert, die Anwendung in Kachel 1 jedoch keine Kenntnis von einer weiteren zu programmierenden Interrupt-Steuerung in Kachel 2 hat, muss die Programmierung

der lokalen Interrupt-Steuerung in Kachel 2, stellvertretend für den Prozessor in Kachel 1, von der Systemsteuerung gemacht werden.

Wenn nun die Anwendung in Kachel 1 eine kritische Anwendung ist und die Anwendung in Kachel 2 eine weniger kritische Anwendung, womit diese auch niedrigere Anforderung erfüllen muss und somit das Risiko für eine Ausfall oder ein nicht spezifiziertes Verhalten höher ist, kann der Mindestabstand zwischen aufeinanderfolgenden Nachrichten von Aufgabe T2 an T1 durch die Interrupt-Überwachung in der Netzwerkschnittstelle von Kachel 2 überwacht werden. In diesem Fall programmiert die Systemsteuerung den Minimalabstand für den entsprechenden Interrupt, welcher auch für die Analyse verwendet wurde, sowie, wie in Abschnitt 4.2.10 beschrieben, eine automatische Reaktion für den Fall, dass der Minimalabstand unterschritten wird. Danach ist das System fertig konfiguriert und die Anwendungen in den beiden Kacheln können gestartet werden.

Die Anwendung in Kachel 2 programmiert zunächst einen Zeitgeber, welcher Aufgabe T2 periodisch aktiviert (1). Damit dies möglich ist, muss der entsprechende Interrupt zunächst in der lokalen Interrupt-Steuerung demaskiert werden. Wenn Aufgabe T2 aktiviert wird, schreibt sie eine neue Nachricht für die Anwendung in Kachel 1 in ihren lokalen Speicher (2) und löst daraufhin den zugehörigen Interrupt aus (3). Dieser wird, wie zuvor durch die Systemsteuerung programmiert, an die Netzwerkschnittstelle geleitet (4). Von dort wird dieser, sofern der zuvor programmierte Mindestabstand zu dem vorherigen Interrupt eingehalten wurde, entsprechend der Interrupt-Übersetzungstabelle als Paket an Kachel 1 gesendet (5). Das Paket wird in der Netzwerkschnittstelle von Kachel 1 entpackt und die zugehörige Interrupt-Leitung, wie von einem herkömmlichen lokalen Peripheriemodul, aktiviert (6). Die lokale Interrupt-Steuerung leitet den zuvor demaskierten Interrupt an den lokalen Prozessor weiter (7). Nachdem Aufgabe T1 durch den empfangenen Interrupt aktiviert wurde, liest T1 die neue Nachricht aus dem lokalen Adressbereich, in welchem der gemeinsam verwendete Speicher für den Nachrichtenaustausch erwartet wird, sich hier jedoch die Netzwerkschnittstelle befindet (8). Die Adressübersetzung in der Netzwerkschnittstelle der Kachel 1 übersetzt den lokalen Zugriff in einen Zugriff über das NoC auf den Speicher in Kachel 2. Sobald das Antwortpaket mit der neuen Nachricht von Kachel 2 in der Netzwerkschnittstelle von Kachel 1 eintrifft, wird es entpackt und als Antwort auf den lokalen Lesezugriff an Aufgabe T1 geliefert (9).

Sollte ein Fehler in einer der Kacheln erkannt werden, beispielsweise durch einen unterschrittenen Minimalabstand zwischen zwei aufeinanderfolgenden Interrupts von Aufgabe T2 an Aufgabe T1, kann die betroffene Anwendung leicht auf eine andere Kachel migriert werden, ohne dass die Anwendung selbst hierfür angepasst werden muss. Die Systemsteuerung muss lediglich den Interrupt- und Adressübersetzungsmechanismus und eventuell die Überwachungsmechanismen in der neuen Kachel programmieren und die betroffenen Anwendung dort neu starten.

4.7. Zusammenfassung

In diesem Kapitel wurden der Gesamtaufbau sowie die einzelnen Komponenten der IDAMC-Plattform im Detail beschrieben. Der Schwerpunkt lag hierbei auf den Netzwerkschnittstellen, welche die Kacheln und das NoC verbinden. In den Netzwerkschnittstellen sind unter anderem die Hardware-Mechanismen für eine neuartige Form der Virtualisierung für Vielkernplattformen sowie für die Separierung von unterschiedlich kritischen Anwendungen auf einer gemeinsamen Plattform implementiert. Die Virtualisierung wird über transparente und flexible Adress- und Interrupt-Übersetzungsmechanismen realisiert und erlaubt die sichere Integration von bestehenden

Anwendung sowie eine Änderung der Zuordnung von Ressourcen zur Laufzeit ohne Anpassung der Gastsysteme. Die Separierung von unterschiedlich kritischen Anwendungen wird über eine Laufzeitüberwachung von gemeinsam verwendeten verteilten Komponenten, von kachelübergreifender Kommunikation sowie des Energieverbrauchs innerhalb der Kacheln und von diesen ausgehend sichergestellt. Weiterhin wurde die Steuerung des Gesamtsystems durch eine zentrale Systemsteuerung sowie ein ausführliches Beispiel für die Kommunikation zwischen Anwendungen auf unterschiedlichen Kacheln beschrieben.

5 Evaluierung

Im folgenden Kapitel wird die Evaluierung der einzelnen Mechanismen zur Laufzeitüberwachung unterschiedlich kritischer Anwendungen aus Kapitel 4 beschrieben. Hierbei wird auch die für die Überwachung des Energieverbrauchs notwendige Charakterisierung von energieintensiven Ereignissen erläutert. Weiterhin werden Syntheseergebnisse von zwei unterschiedlichen Beispielkonfigurationen präsentiert sowie die Plattform und deren Mechanismen mit Ansätzen aus anderen Arbeiten, welche vor allem in Kapitel 2 beschrieben wurden, verglichen.

Um die Mechanismen zur flexiblen und transparenten Zuordnung von Anwendungen, zur Laufzeitüberwachung sowie die IDAMC-Plattform insgesamt zu evaluieren, wie bereits teilweise in [3, 2, 1] veröffentlicht, wurde das System auf der HAPS-62-Plattform [83] von Synopsys implementiert. Die HAPS-62-Plattform gehört zu einer Serie von Produkten, um die Funktionalität einer ASIC-Neuentwicklung zunächst auf einem Prototypen evaluieren zu können. Abbildung 5.1 zeigt ein Blockdiagramm der HAPS-62-Plattform mit den zentralen Komponenten, zwei Virtex-6-LX760-FPGAs [94] von Xilinx. Weitere Varianten mit einem oder auch vier FPGAs existieren ebenfalls in der HAPS-Serie. Neben den FPGAs, Tastern und Leuchtdioden, existieren auf der Platine eine Reihe von Steckern. Diese können verwendet werden, um die Plattform mit zusätzlichen Platinen mit externen Speicherschnittstellen, mit Peripheriemodulen oder auch mit zusätzlichen Verbindungen zwischen den FPGAs erweitern zu können. Weiterhin können die Stecker dafür verwendet werden, mehrere HAPS-Systeme zu einer größeren Plattform zusammenzustecken. Für das IDAMC-System wurde die HAPS-62-Plattform zunächst mit einem DDR2-Modul und einem Modul mit weiteren Tastern, Leuchtdioden und Siebensegmentanzeigen erweitert.

Kleine und mittlere Konfiguration der IDAMC-Plattform benötigen lediglich einen der verfügbaren FPGAs des HAPS-62-Systems und damit auch keine Verbindungen zwischen den FPGAs. Größere Konfigurationen können relativ leicht auf mehrere FPGAs einer Platine oder sogar auf mehrere HAPS-Systeme aufgeteilt werden. Hierbei sollten jeweils komplette Kacheln und Switches auf einem FPGA platziert werden und die Verbindungen zwischen den Switches für Verbindungen zwischen den FPGAs verwendet werden. Sehr große Systemen, bei denen es schwierig ist, ein korrektes Zeitverhalten zu erreichen, können auch als GALS-System implementiert werden, indem man die Ein- und Ausgangspuffer der Netzwerkschnittstellen für die Übergänge zwischen den Taktdomänen einsetzt.

Die Implementierung der IDAMC-Plattform ist jedoch nicht auf HAPS beschränkt. Andere Systeme, auch mit anderen FPGAs, können ebenfalls eingesetzt werden, was jedoch erfordern kann, dass technologiespezifische Komponenten, zum Beispiel zur Ansteuerung des externen DDR2-Speichers, ersetzt werden müssen. Die auf der HAPS-Plattform eingesetzten Virtex-6-LX760-FPGAs waren zum Zeitpunkt der Entwicklung der IDAMC-Plattform die größten kommerziell verfügbaren FPGAs. Die Implementierung auf kleineren FPGAs kann daher zu Einschränkungen bei der Maximalanzahl an Kacheln und deren Konfiguration führen.

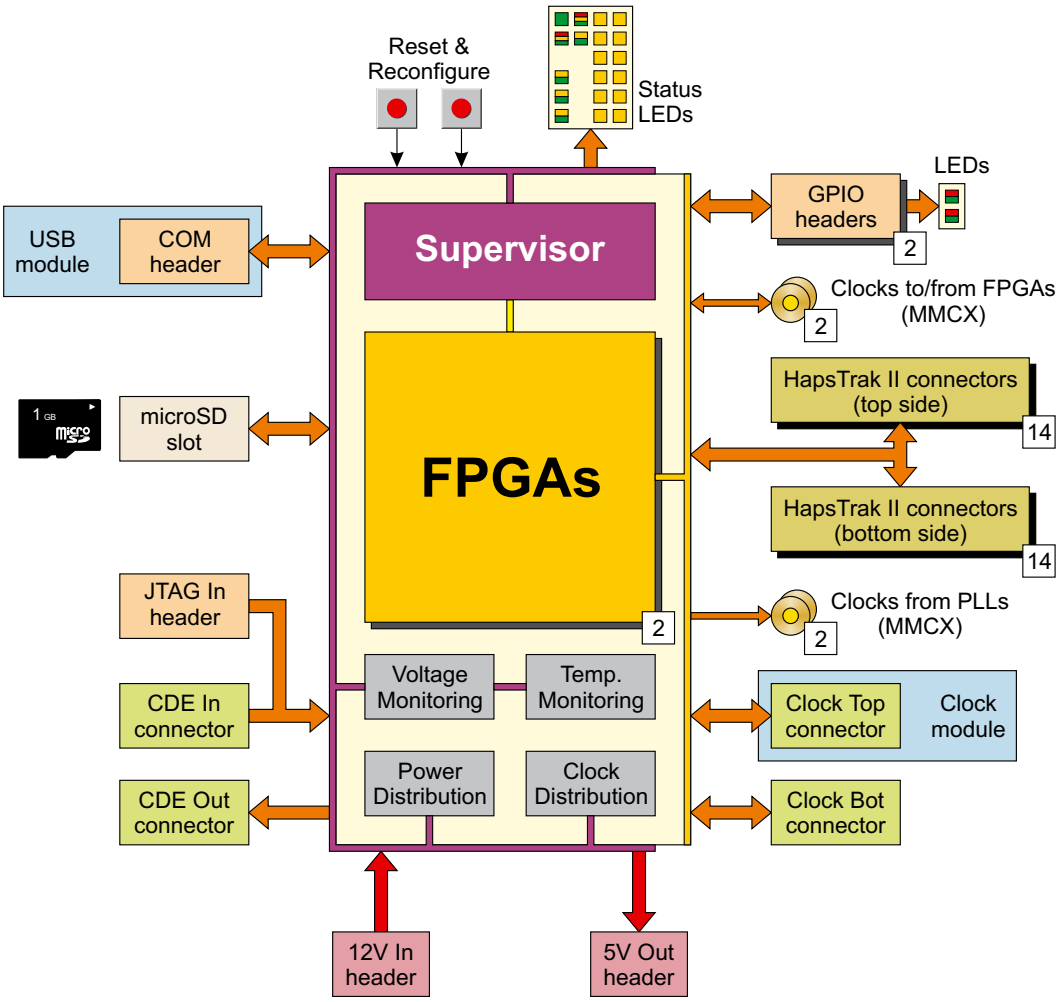

Abbildung 5.1.: HAPS-62-Blockdiagramm [83]

5.1. Gemeinsam verwendete Komponenten

Da vor allem die Charakterisierung von energieintensiven Ereignissen auf Gatterebene, welche in Abschnitt 5.2.1 beschrieben wird, bei größeren Konfigurationen zu erheblichen Synthese- und Simulationszeiten führte, wurde zunächst eine Minimalkonfiguration implementiert. Aufgrund der Skalierbarkeit der in Kapitel 4 beschriebenen Mechanismen weist diese Konfiguration dennoch alle Charakteristiken des Systems auf, um so eine aussagekräftige Evaluation durchführen zu können. Die Konfiguration wird im Folgenden Minimalsystem genannt, auch wenn noch kleinere Konfigurationen, beispielsweise mit nur einer einzelnen Kachel, möglich sind, welche jedoch keine ausreichenden Evaluation der beschriebenen Mechanismen ermöglichen. Eine größere Konfiguration der IDAMC-Plattform mit weiteren Kacheln und Switches wird in Abschnitt 5.3 vorgestellt.

Das für die Experimente in diesem und dem nächsten Abschnitt verwendete Minimalsystem besteht, wie in Abbildung 5.2 dargestellt, aus vier Kacheln in einer Zwei-mal-Zwei-Anordnung, wobei jede Kachel an einen separaten Switch angeschlossen ist. Die Systemsteuerung auf der Kachel mit der Kachelidentifikationsnummer 0, unten links in Abbildung 5.2, besteht aus einem LEON3-Prozessor, einer Debug-Einheit mit zugehöriger Debug-Schnittstelle, einer UART-Schnittstelle für

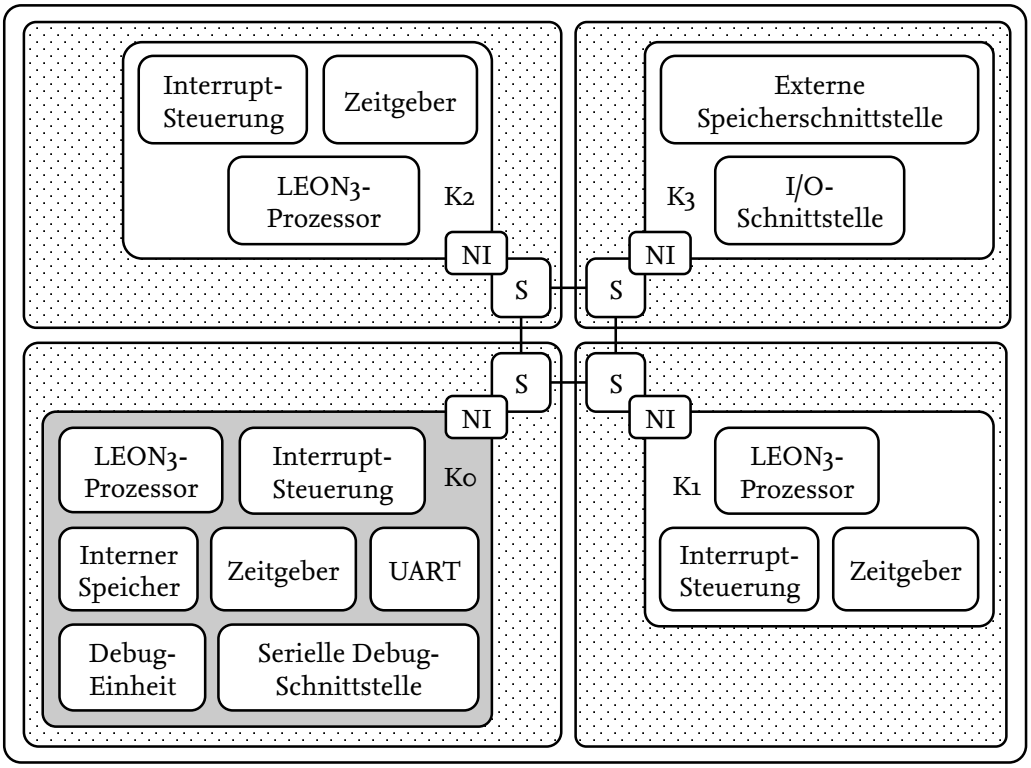

Abbildung 5.2.: IDAMC-Minimalsystem

Textausgaben an eine Konsole, einer lokale Interrupt-Steuerung, einem Zeitgeber sowie 256 Kilobytes internem Speicher. Weitere vorhandene Komponenten, wie eine AHB/APB-Brücke sowie die AHB-Steuerung, sind der Übersichtlichkeit halber in Abbildung 5.2 und Abbildung 5.8 weiter unten nicht abgebildet. Von der Systemsteuerung aus werden Kacheln und verteilte Ressourcen an Anwendungen zugewiesen und deren Separierung über die Programmierung der dezentralen Laufzeitüberwachungsmechanismen in den Netzwerkschnittstellen sichergestellt.

Die Kacheln K1 und K2 sind identische Rechenkacheln jedoch ohne eigenen lokalen Speicher. Beide Rechenkacheln bestehen aus einem einzelnen LEON3-Prozessor, einem Zeitgeber sowie einer Interrupt-Steuerung. Alle Prozessoren des Systems beinhalten jeweils einen Daten- sowie einen Befehls-Cache mit einer Größe von jeweils einem Kilobyte. Die Speicherverwaltungseinheit (MMU) und die Fließkommaeinheit (FPU) sind in allen Prozessoren über die zentrale Konfigurationsdatei deaktiviert worden. Die Adressübersetzungen in den Netzwerkschnittstellen der Kacheln K0, K1 und K2 sind, wie in Abschnitt 4.2.8 beschrieben, so konfiguriert, dass sie pro Kachel jeweils 16 Speicher- und 16 I/O-Adressbereiche mit zugehöriger Laufzeitüberwachung unterstützen. Die Kachel K3 ist als kombinierte Speicher- und Peripheriekachel konfiguriert und umfasst lediglich eine Speicherschnittstelle, um den externen DDR2-Speicher ansprechen zu können, sowie eine I/O-Schnittstelle für die Ansteuerung des externen Moduls mit den Siebensegmentanzeigen, Tastern und Leuchtdioden. Kachel K3 beinhaltet keinen lokalen Prozessor und damit auch keine Adressübersetzung in der zugehörigen Netzwerkschnittstelle.

5.1.1. Experimente

Die folgenden Experimente wurden auf dem oben beschriebenen Minimalsystem implementiert, um zunächst zu demonstrieren, wie eine Abweichung des Verhaltens einer weniger kritischen oder nicht sicherheitsrelevanten Anwendung eine höher kritische Anwendung bei der Verwendung einer gemeinsamen Komponente negativ beeinflussen kann. Anschließend wird gezeigt, wie der Mechanismus zur Überwachung von gemeinsam verwendeten Komponenten zur Laufzeit, welcher in Abschnitt 4.2.10 beschrieben wird, eine ausreichende Separierung zwischen unterschiedlich kritischen Anwendungen sicherstellt, um so eine unabhängige Entwicklung und Zertifizierung zu ermöglichen.

In den Experimenten führen die Rechenkacheln K1 und K2 die eigentlichen Anwendungen aus, welche jeweils auf dem Dhrystone-Benchmark [90] basieren. Eine der Anwendungen wird dabei als kritisch und eine als nicht kritisch angenommen. Beide Rechenkacheln verfügen über keinen lokalen Speicher, so dass sich sowohl die Daten- als auch die Programmspeicher der Anwendungen auf den Rechenkacheln physisch in dem gemeinsam verwendeten Speicher in Kachel K3 befinden. Die Systemsteuerung auf der Kachel mit der internen Bezeichnung K0 weist den Anwendungen jeweils eine Kachel zu, programmiert die Adressübersetzungstabellen und die Laufzeitüberwachung in den Netzwerkschnittstellen. Nach der Aktivierung der Rechenkacheln liest die Systemsteuerung die überwachten Daten periodisch aus, um diese zur Auswertung und Darstellung über die UART-Schnittstelle an den Steuerrechner zu schicken.

Die Systemsteuerung blendet zunächst über die Adressübersetzungstabellen jeweils zwei größere Speicheradressbereiche von jeweils einem Megabyte lokal in den Rechenkacheln ein, einen Adressbereich an der Adresse, an der in den Rechenkacheln der Daten- und Programmspeicher erwartet wird und einen Adressbereich an der Adresse, an der ein Festwertspeicher für den Startprozess erwartet wird. Hierbei werden Zugriffe auf den Adressbereich für den Startprozess in beiden Rechenkacheln auf denselben Speicherbereich in Kachel K3 übersetzt. Dieser Speicherbereich wird also von den Anwendungen in beiden Kacheln gemeinsam verwendet, ist jedoch in beiden Adressübersetzungstabellen mit einem Schreibschutz versehen. Lokale Zugriffe auf den Programm- und Datenspeicher werden ebenfalls auf denselben Speicher, aber auf getrennte Speicherbereiche in Kachel K3 übersetzt. Diese Speicherbereiche sind damit logisch voneinander getrennt. Zugriffe von den beiden Anwendungen können sich aber immer noch zeitlich Beeinflussen, da hierbei die zugehörige Speicherschnittstelle trotzdem gemeinsam verwendet wird. Da sich die Zugriffe auf den gemeinsamen Speicher bereits im NoC beeinflussen können, werden auch die Verbindungen zwischen den beiden Rechenkacheln und der Kachel K3 über unterschiedliche virtuelle Kanäle logisch voneinander separiert.

Für die Experimente wird angenommen, dass Kachel K1 eine kritische Aufgabe T1 ausführt und Kachel K2 ein nicht kritische Aufgabe T2. Beide Aufgaben werden durch ihre lokalen Zeitgeber alle $4,5\,ms$ periodisch aktiviert. Die Frist, zu der T1 spätestens abgeschlossen sein muss, liegt bei $3,8\,ms$ nach der Aktivierung. Indem, wie in Abbildung 5.3 oben dargestellt, T1 zunächst exklusiv auf der Plattform ausgeführt wird, kann für den Fall ohne Beeinflussung durch weitere Aufgaben und unter Verwendung der Laufzeitüberwachung eine späteste Antwortzeit von $3,357\,ms$ ermittelt werden.

Die maximale Anzahl an Lese- und Schreibzugriffen auf den gemeinsam verwendeten Speicher durch die nicht kritische Aufgabe T2 innerhalb der Periode der kritischen Aufgabe kann ebenfalls

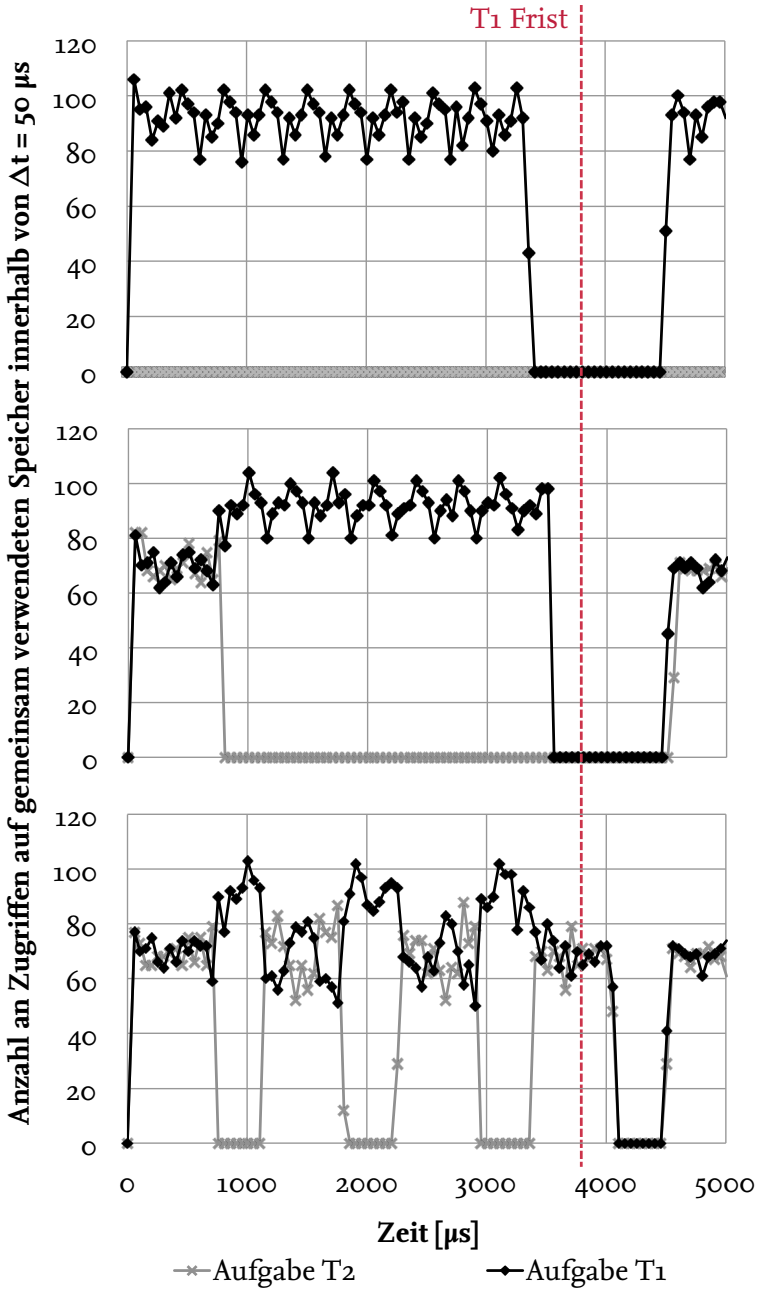

Abbildung 5.3.: Verwendung eines gemeinsamen Speichers

über die Laufzeitüberwachung ermittelt werden, indem hierfür die Aufgabe T2 exklusiv auf der Plattform ausgeführt wird. In $4,5\,ms$ liest T2 maximal 902 Wörter aus dem gemeinsam verwendeten Speicher und schreibt maximal 85 Wörter. Hierbei benötigt ein Lesezugriff $400\,ns$ und ein Schreibzugriff $100\,ns$ pro gelesenem beziehungsweise pro geschriebenem Datenwort. Bei der Analyse des Zeitverhaltens von unterschiedlich kritischen Aufgaben unter Berücksichtigung von gemeinsam verwendeten Komponenten wird, wie in Abschnitt 3.1 beschrieben, zur Vereinfachung nicht zwischen Lese- und Schreibzugriffen getrennt. Da der zeitliche Unterschied auf der realen IDAMC-Plattform jedoch nicht zu vernachlässigen ist, werden hier Lese- und Schreibzugriffe separat gezählt, um die Überabschätzung der Analyse zu reduzieren. Mit Gleichung 3.1 kann die maximale zusätzliche Verzögerung durch die gemeinsame Verwendung des Speichers in dem Experiment über

$$D_{T_1,S} = \tilde{\eta}_{T_2 \to S}(4,5\,ms) \cdot m = 902 \cdot 400\,ns + 85 \cdot 100\,ns = 369,3\,\mu s \tag{5.1}$$

bestimmt werden.

In Abbildung 5.3 Mitte kann man erkennen, wie die Anzahl an Zugriffen durch die kritische Aufgabe T1, im Vergleich zu Abbildung 5.3 oben, in dem Zeitraum, in dem auch die nicht kritische Aufgabe auf den gemeinsamen Speicher zugreift, verringert ist. Weiterhin sieht man, wie hiermit die Antwortzeit der kritischen Aufgabe T1 näher an ihre Frist rückt, welche in der Abbildung als rot gestrichelte Linie dargestellt ist. Mit der ermittelten Antwortzeit während einer exklusiven Nutzung des Speichers von $3,357\,ms$ und der berechneten maximalen zusätzlichen Verzögerung durch die gemeinsame Nutzung von $369,3\,\mu s$ ergibt sich eine neue späteste Antwortzeit für die kritische Aufgabe T1 von $3,726\,ms$. Solange sich also die nicht kritische Aufgabe wie erwartet verhält, liegt die späteste Antwortzeit von T1 vor der Frist und das System ist damit sicher ausführbar.

Die gemessene Antwortzeit von T1 liegt in dem Experiment bei lediglich $3,507\,ms$. Der Unterschied zwischen der analytischen Lösung und der gemessenen Zeit liegt an der pessimistischen Abschätzung der Beeinflussung durch die Zugriffe der nicht kritischen Aufgabe T2. Es wird hierbei angenommen, dass jeder Zugriff durch T2 die Ausführung der kritischen Aufgabe um die komplette Zugriffsdauer verzögert, was normalerweise nicht der Fall ist. Manche Zugriffe von T2 führen zu keiner oder nur teilweise zu einer Verzögerung der Aufgabe T1.

Da die nicht kritische Aufgabe T2 jedoch nicht dieselben strengen Anforderungen wie die kritische Aufgabe T1 erfüllt, kann selbst die obige pessimistische Abschätzung des Verhaltens von T2 zur Laufzeit nicht als sicher angenommen werden. Mit einem theoretischen Maximum von zwei Zugriffen pro Mikrosekunde auf den gemeinsam verwendeten Speicher durch die nicht kritische Aufgabe T2, welches schon in dem Beispiel in Abschnitt 3.4 angenommen wurde, ergibt sich ein theoretisches Maximum für die Anzahl der Zugriffe durch eine fehlerhafte Aufgabe T2 innerhalb einer Periode von Aufgabe T1 von $2/\mu s \cdot 4,5\,ms = 9000$. Dies würde im schlechtesten Fall also zu einer Verzögerung von $9000 \cdot 400\,ns = 3,6\,ms$ führen, die bei der Bestimmung der spätesten Antwortzeit der kritischen Aufgabe T1 berücksichtigt werden müsste. Hiermit könnte die späteste Antwortzeit von T1 durch eine fehlerhafte Aufgabe T2 weit hinter der Frist liegen und T1 damit ebenfalls fehlschlagen. Ohne einen Separierungsmechanismus wäre die obige Konfiguration also, solange die nicht kritische Aufgabe T2 nicht dieselben hohen Anforderungen wie die kritische Aufgabe T1 erfüllt, nicht sicher zeitlich ausführbar.

Eine Verletzung der Frist von Aufgabe T1 durch eine fehlerhafte zusätzliche Verzögerung kann durch zusätzliche Aktivierungen der Aufgabe T2, mehr Speicherzugriffen von Aufgabe T2 pro

Aktivierung oder einer Kombination aus beidem ausgelöst werden. Dies kann beispielsweise durch ein gekipptes Bit in der Zeitbasis des lokalen Zeitgebers der Kachel von T2 oder durch einen abweichenden Programmfluss aufgrund von unerwarteten Eingangsdaten hervorgerufen werden. Abbildung 5.3 unten zeigt den Fall, in dem eine fehlerhafte Zeitbasis des lokalen Zeitgebers in Kachel K2 zu vier anstatt einer Aktivierung der nicht kritischen Aufgabe T2 innerhalb einer Periode der kritischen Aufgabe T1 führt. Die Antwortzeit beträgt in dem dargestellten Fall $4,057\,ms$. T1 verpasst also die Frist und schlägt fehl.

Mit Gleichung 3.1 und Gleichung 3.2 kann in diesem Experiment, wie in dem Beispiel in Abschnitt 3.1 gezeigt, eine obere Grenze von 185 zusätzlichen, über die erwartete Anzahl hinausgehenden Zugriffen von Aufgabe T2 auf den gemeinsam verwendeten Speicher ermittelt werden, damit die Frist der kritischen Aufgabe T1 gerade noch garantiert werden kann. Mit einem theoretischen Maximum von zwei Zugriffen pro Mikrosekunde auf den gemeinsamen Speicher durch Aufgabe T2 ergibt sich eine maximale Reaktionszeit von $\frac{185}{2/\mu s} = 92,5\,\mu s$, welche für eine sichere Separierung einzuhalten ist. Um eine geeignete Reaktion innerhalb der Prozesssicherheitszeit abschließen zu können, muss also ein erhöhtes Aufkommen an Zugriffen durch Aufgabe T2 auf den gemeinsam verwendeten Speicher schneller erkannt und weitere Zugriffe verhindert werden als die kürzeste Zeit, die Aufgabe T2 benötigt, um 185 mal auf den gemeinsamen Speicher zuzugreifen.

Da die Laufzeitüberwachung gemeinsam verwendeter Komponenten, welche in Abschnitt 4.2.10 beschrieben wird, bereits den ersten über die erwartete Anzahl hinausgehenden Zugriff erkennen und verhindern kann, lassen sich auf der IDAMC-Plattform auch Aufgaben mit deutlich strengeren Anforderungen als in diesem Experiment sicher separieren.

Abbildung 5.4 zeigt die Antwortzeit der kritischen Aufgabe T1 in Abhängigkeit von der Anzahl an Aktivierung der nicht kritischen Aufgabe T2 innerhalb einer Periode von T1, einmal mit aktivierter Laufzeitüberwachung und einmal ohne. Die Laufzeitüberwachung der Kachel K2 ist in dem Experiment von der Systemsteuerung so programmiert worden, dass Kachel K2 deaktiviert wird, falls eine Abweichung vom erwarteten Verhalten erkannt wird. Von einer Abweichung wird hier gesprochen, wenn Aufgabe T2 innerhalb von $4,5\,ms$ öfter als 902 mal von dem gemeinsam verwendeten Speicher liest beziehungsweise öfter als 85 mal schreibend auf den Speicher zugreift, was einer einmaligen Aktivierung von Aufgabe T2 innerhalb der Periode von T1 entspricht. Abbildung 5.4 zeigt, dass die Antwortzeit von T1 mit der Anzahl an Aktivierung von T2 ohne aktivierte Laufzeitüberwachung proportional ansteigt und die Frist der kritischen Aufgabe T1 für mehr als zwei Aktivierungen von T2 überschritten wird. Mit aktivierter Laufzeitüberwachung werden alle Zugriff von Aufgabe T2 auf den gemeinsam verwendeten Speicher, die über die erwartete Anzahl hinausgehen, unterbunden und die Antwortzeit von T1 bleibt, unabhängig von der erhöhten Anzahl an Aktivierungen von Aufgabe T2, konstant.

Das Experiment zeigt, dass die Verwendung von gemeinsamen Komponenten durch unterschiedlich kritische Anwendungen durch die in Abschnitt 4.2.10 beschriebene Laufzeitüberwachung sicher separiert werden kann. Die Separierung wird sichergestellt, indem die Beeinflussung durch eine weniger kritische oder nicht sicherheitsrelevante Aufgabe auf eine höher kritische Aufgabe durch eine sicher obere Grenze eingeschränkt wird.

In Abbildung 5.4 liegt die Antwortzeit mit aktivierter Laufzeitüberwachung deutlich vor der Frist. Durch die schnelle Reaktionszeit der verwendeten Laufzeitüberwachung könnte für die nicht kriti-

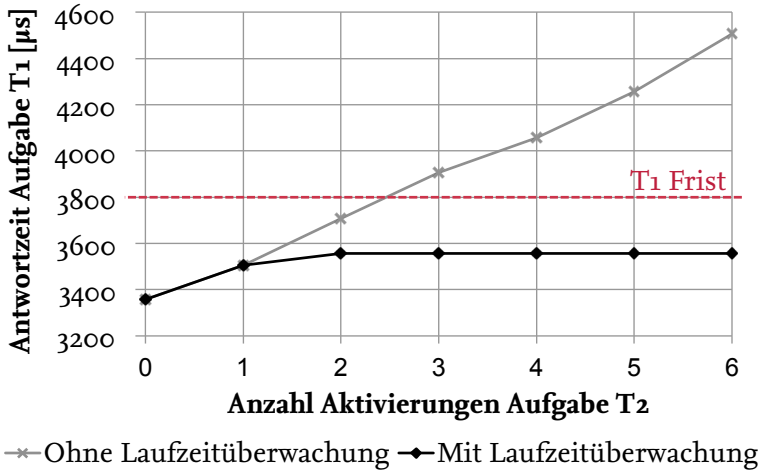

Abbildung 5.4.: Antwortzeit mit und ohne Laufzeitüberwachung

Ressourcen	IDAMC	NoC	K 0	K 1/2	K 3	NI 0/1/2	NI 3	Überwachung gemeinsame Komponenten 0/1/2
#Flipflops	40335 (4%)	18820	5252	4507	2537	2865	2086	639
#LUTs	64974 (13%)	24910	12068	9889	4673	6168	3774	1390
#BRAMs	118 (12%)	12	79	12	3	6	3	2

Tabelle 5.1.: Syntheseergebnisse Zwei-mal-Zwei-System

sche Aufgabe T2 in dem Experiment auch, wie in Abschnitt 3.4 beschrieben, eine gewisse Abweichung vom erwarteten Verhalten akzeptiert werden, solange T1 ihre Frist einhalten kann.

5.1.2. Implementierung

Das oben beschriebene Minimalsystem wurde auf einem der beiden verfügbaren FPGAs der HAPS-62-Plattform implementiert. Das gesamte IDAMC-System wird dabei, mit Ausnahme der DDR2-Schnittstelle, mit 80 MHz getaktet. Die DDR2-Schnittstelle wird mit der doppelten Frequenz betrieben. Die Ergebnisse der Synthese sind in Tabelle 5.1 dargestellt. Die erste Spalte zeigt die Ressourcen, welche auf einem FPGA hauptsächlich vorkommen, Flipflops zum Speichern einzelner Bits, programmierbare Wahrheitstabellen (engl. *Lookup Tables (LUTs)*), sowie Speicherblöcke (engl. *Block Random Access Memorys (BRAMs)*). Die weiteren Spalten zeigen den Verbrauch der jeweiligen Ressourcen für das Gesamtsystem, das NoC, die Kacheln K0-K3, wobei der Ressourcenverbrauch für die Kachel K1 und K2 identisch ist, die Netzwerkschnittstellen, wobei hier der Ressourcenverbrauch der Netzwerkschnittstellen NI0-2 der Kacheln K1-3 identisch ist, sowie den Ressourcenverbrauch des Laufzeitüberwachungsmechanismus für gemeinsam verwendete Komponenten pro Kachel. Die Gesamtauslastung eines der Virtex-6-LX760-FPGA ist in Klammern hinter den Zahlen des Gesamtressourcenverbrauchs der IDAMC-Minimalkonfiguration dargestellt.

Sowohl die Frequenz als auch der Ressourcenverbrauch für die einzelnen Kacheln sind vergleichbar mit anderen auf der GRLIB-Bibliothek basierenden Systemen. Die Anzahl der verwendeten BRAMs in den Netzwerkschnittstellen hängt eng mit der Anzahl möglicher Speicher- und I/O-Adressbereiche

für verteilte Komponenten zusammen. Diese Anzahl bestimmt die Größe der Adressübersetzungstabellen sowie die Größe der Tabellen für die Überwachung von gemeinsam verwendeten Komponenten. Da Kachel K3 keinen lokalen AHB-Master beinhaltet, ist der Ressourcenverbrauch der zugehörigen Netzwerkschnittstelle NI3 geringer als der Verbrauch der anderen Netzwerkschnittstellen. Die Netzwerkschnittstelle NI3 benötigt in der implementierten Konfiguration keine Slave-Schnittstelle, keine Adressübersetzung und auch keine Laufzeitüberwachung von gemeinsam verwendeten Komponenten.

Der Laufzeitüberwachungsmechanismus zur Überwachung von jeweils 16 verteilten Speicher- und 16 I/O-Adressbereichen pro Kachel K0-K2 erhöht den Ressourcenverbrauch des Minimalsystems um etwa sechs Prozent. Jeder weitere Adressblock erhöht den Speicherverbrauch, jedoch nicht die Anzahl an Flipflops und LUTs, welche auch für sehr große Systeme mit vielen verteilten Komponenten konstant bleiben. Erhöhte Speicheranforderungen werden erst dann sichtbar, wenn die Speicheranforderungen die Kapazität der aktuell verwendeten BRAMs überschreiten.

5.2. Energieverbrauch

Um die Mechanismen zur Separierung des Energieverbrauchs von unterschiedlich kritischen Anwendungen zu evaluieren, wurde dasselbe Minimalsystem wie für die Evaluierung der Mechanismen zur Separierung gemeinsam verwendeter Komponenten eingesetzt. Für die im Folgenden beschriebenen Experimente wurde jedoch die externe Speicherschnittstelle in Kachel K3 durch einen zwei Megabyte großen internen Speicher ersetzt, um den Fokus der Experimente mehr auf den Energieverbrauch auf dem Chip zu legen.

5.2.1. Charakterisierung

Um den Energieverbrauch von einzelnen Anwendungen sowie die Energiedichte in einzelnen Chip-Regionen der IDAMC-Plattform über das Zählen von Ereignissen abschätzen zu können, müssen alle Ereignisse des Systems, welche einen signifikanten Anteil am Energieverbrauch haben, charakterisiert werden. Hierfür können Mikrobenchmarks und lineare Regression, wie in [16, 18] beschrieben, eingesetzt werden. Die verwendeten Mikrobenchmarks werden auf dem Minimalsystem auf dem Prozessor in Kachel K0 ausgeführt und enthalten eine variable Anzahl an Operation derselben Art. Ein Mikrobenchmark besteht beispielsweise ausschließlich aus einer variablen Anzahl an Lesezugriffen auf den internen Speicher in Kachel K3, um so die verursachte dynamische Verlustleistung in den Netzwerkschnittstellen, den Switches auf dem Pfad von Kachel K0 nach K3 und dem internen Speicher selbst pro gelesenem Wort bestimmen zu können.

Die Mikrobenchmarks werden in einer Simulation auf Gatterebene ausgeführt. Hierbei wird die Schaltaktivität (engl. *Switching Activity*) in den bereits platzierten und verbundenen Elementen der FPGA-Bibliothek aufgezeichnet. Der Ansatz ist jedoch nicht auf FPGAs begrenzt. Anstelle der FPGA-Bibliothek kann hier auch eine Standardzellenbibliothek wie in [18] verwendet werden. Die aufgezeichnete Schaltaktivität wird anschließend mit Xilinx' XPower Analyzer analysiert, um, wie in [18] beschrieben, die dynamische Verlustleistung in den einzelnen Komponenten zu bestimmen.

Abbildung 5.5 verdeutlicht den linearen Zusammenhang zwischen der ermittelten dynamischen Verlustleistung und der bekannten Anzahl und Art an Ereignissen, welche durch die jeweiligen Mikrobenchmarks hervorgerufen werden. Mittels linearer Regression lassen sich, wie in [39], die

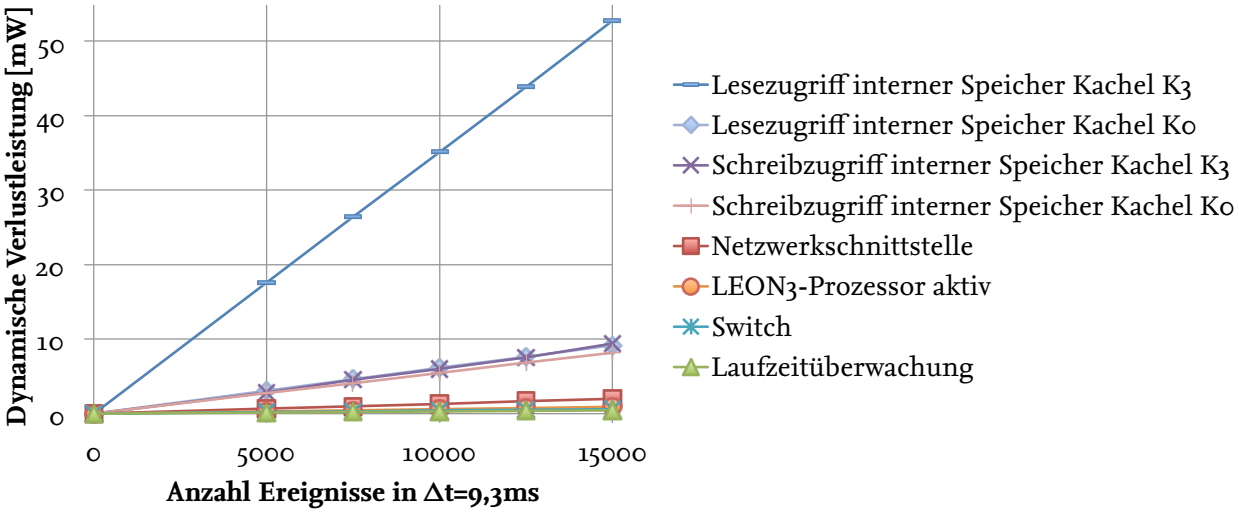

Abbildung 5.5.: Zusammenhang Anzahl Ereignisse und Verlustleistung

Ereignis	Energiegewichtung [nJ]
Lesezugriff interner Speicher Kachel K3	32.66
Lesezugriff interner Speicher Kachel K0	5.716
Schreibzugriff interner Speicher Kachel K3	5.62
Schreibzugriff interner Speicher Kachel K0	5.08
Netzwerkschnittstelle	1.24
LEON3-Prozessor aktiv	0.59
Switch	0.39
Laufzeitüberwachung	0.27

Tabelle 5.2.: Energiegewichtungen

Steigungen der Geraden ermitteln, welche dem Zuwachs an dynamischer Verlustleistung pro Ereignis entsprechen.

Mit den Verlustleistungen pro Ereignis und der bekannten Laufzeit der Mikrobenchmarks lassen sich die Energiegewichtungen der jeweiligen Ereignisse, welche in Tabelle 5.2 für die wichtigsten Ereignisse aufgeführt sind, bestimmen. Die Energiegewichtung eines Ereignisses, wie das Lesen des internen Speichers in Kachel K3, beinhaltet stets die Energie der gesamten Operation, also auch die Energie, welche in den beiden Netzwerkschnittstellen, den drei Switches und der Laufzeitüberwachung auf dem Weg von Kachel K0 nach K3 verbraucht wird.

Da das Lesen des relativ großen internen Speichers in Kachel K3 deutlich mehr Energie benötigt als die verschiedenen Operationen innerhalb des LEON3-Prozessors, wird hier lediglich eine einzelne Energiegewichtung für alle internen Operationen des Prozessors verwendet. Die Reduzierung der Anzahl von unterschiedlichen Ereignissen führt zu einer stark verminderten Komplexität der Laufzeitüberwachung und hat dabei nur wenig Auswirkungen auf die Genauigkeit der Abschätzung. Hierbei ist nur wichtig, dass von den ermittelte Energiegewichtungen der einzelnen Operationen der größte Wert für die Energieabschätzung verwendet wird, um eine sichere obere Grenze für den Energieverbrauch der einzelnen Anwendungen zu erhalten.

Aufgabe	Kritikalität	Aktivierungs-intervall	Laufzeit	Speicherzugriffe K3 pro Aktivierung		Aktive Prozessorzyklen pro Aktivierung
				Lesen	Schreiben	
T1	hoch	$1,6\,ms$	$1,2\,ms$	3337	291	2625
T2	niedrig	$2,4\,ms$	$720\,\mu s$	2107	180	1640

Tabelle 5.3.: Experimentelles Setup

Andere Konfigurationen der IDAMC-Plattform mit unterschiedlichen Komponenten, Speicher- oder Cache-Größen erfordern zwar einen zusätzlichen einmaligen Aufwand, um geeignete Ereignisse auszuwählen und zu charakterisieren, jedoch keine grundsätzliche Änderung an der verwendeten Methode. Darüber hinaus ist es möglich, mit bekannten Energiegewichtungen und deren Zusammensetzung, Energiegewichtungen von anderen Konfigurationen zu berechnen. So kann beispielsweise ein Zugriff auf einen weiter entfernten Speicher in einem größeren System als dem vorgestellten Minimalsystem berechnet werden, da die benötigte Energie, um ein Datenwort über einen weiteren Switch zu transportieren, bereits bekannt ist.

Zur Überprüfung der Genauigkeit des Modells wurden zum Abschluss der Charakterisierung komplexere Programme, basierend auf dem Dhrystone-Benchmark [90], ausgeführt, um so die Abweichung der abgeschätzten Verlustleistung von der realen Verlustleistung zu ermitteln. Die Abweichung der Abschätzung gegenüber der tatsächlichen Verlustleistung aus der Netzlistensimulation betrug hierbei zwischen acht und 23 Prozent. Diese Abweichung kann über weitere, feingranularere Ereignisse oder über die Charakterisierung mit einem Programm, welches eine höhere Genauigkeit als Xilinx' XPower Analyzer bietet, verringert werden. In der vorliegenden Arbeit war jedoch nicht eine möglichst hohe Genauigkeit das Ziel, sondern eine sichere Separierung des Energieverbrauchs unterschiedlich kritischer Anwendungen. Da die Abschätzung der dynamischen Verlustleistung stets über der tatsächlichen Verlustleistung lag, ist dieses Ziel hier erreicht worden.

5.2.2. Experimente

In den im Folgenden beschriebenen Experimenten zur Evaluierung des Mechanismus zur Trennung des Energieverbrauchs unterschiedlich kritischer Anwendungen führt Kachel K1, des in Abbildung 5.2 dargestellten Minimalsystems, eine kritische Aufgabe T1 und Kachel K2 eine weniger kritische Aufgabe T2 aus. Beide Aufgaben basieren, wie in den Experimenten für gemeinsam verwendete Komponenten weiter oben, auf dem Dhrystone-Benchmark [90] und verwenden den Speicher in Kachel K3 gemeinsam. Tabelle 5.3 zeigt die Eigenschaften der beiden Aufgaben des Experiments zusammengefasst. Die Laufzeit, die Anzahl an Lese- und Schreibzugriffen auf den gemeinsam verwendeten Speicher in Kachel K3 sowie die aktiven Prozessorzyklen der jeweiligen Aufgaben können vor Beginn des eigentlichen Experiments über die Laufzeitüberwachung ermittelt werden. Tabelle 5.4 zeigt die mit Gleichung 3.7 und Gleichung 3.8 berechneten erwarteten Werte für den Energieverbrauch und die durchschnittliche Verlustleistung im Messintervall von $3,25\,ms$.

Abbildung 5.6 und Abbildung 5.7 zeigen für drei verschiedene Experimente die zur Laufzeit ermittelte dynamische Verlustleistung beziehungsweise den dynamischen Energieverbrauch innerhalb des Messintervalls von $3,25\,ms$. Neben der individuellen Verlustleistung und des Energieverbrauchs der einzelnen Aufgaben, zeigen die Abbildungen zusätzlich die dynamische Gesamtverlustleistung

Aufgabe	Energieverbrauch pro Aktivierung	Energieverbrauch pro Messintervall	Durchschnittliche Verlustleistung im Messintervall
T1	$112,17\,\mu J$	$224,34\,\mu J$	$69,03\,mW$
T2	$70,73\,\mu J$	$141,46\,\mu J$	$43,53\,mW$
	Gesamt	$365,80\,\mu J$	$112,56\,mW$

Tabelle 5.4.: Erwartete Verlustleistung und Energieverbrauch

und den Gesamtenergieverbrauch der beiden Aufgaben T1 und T2 zusammen. Weiterhin wird in Abbildung 5.6 noch die durchschnittliche dynamische Verlustleistung der jeweiligen Aufgaben und deren gemeinsame durchschnittliche Verlustleistung dargestellt und in Abbildung 5.7 die insgesamt im Messintervall noch verfügbare Energie.

In den beiden Abbildungen gehören jeweils die oben, in der Mitte und unten dargestellten Diagramme zu demselben Experiment. In den Abbildungen oben wird der Fall dargestellt, in dem sich beide Aufgaben T1 und T2 wie erwartet verhalten. Da Aufgabe T2 durch ihre geringere Kritikalität lediglich weniger strenge Anforderungen erfüllen muss als die kritische Aufgabe T1, ist das Verhalten von T2 hierbei jedoch nicht garantiert. Es wird nun angenommen, dass, wie in Abbildung 5.7 dargestellt, im Messintervall insgesamt $400\,\mu J$ zur Verfügung stehen. Hiervon verbraucht Aufgabe T1 bei $112,17\,\mu J$ pro Aktivierung und zwei Aktivierungen im Messintervall $224,34\,\mu J$ und Aufgabe T2 mit $70,73\,\mu J$ pro Aktivierung und ebenfalls zwei Aktivierungen $141,46\,\mu J$. Insgesamt werden also, wenn sich das System wie erwartet verhält, in dem betrachteten Intervall $365,8\,\mu J$ verbraucht.

Wenn nun die weniger kritische Aufgabe T2 mehr energieintensive Ereignisse verursacht, beispielsweise durch eine längere Laufzeit pro Aktivierung durch unerwartete Eingangsdaten oder durch eine häufigere Aktivierung durch einen Fehler, zum Beispiel im Zeitgeber, erhöhen sich die Verlustleistung und der Energieverbrauch, wie in den Abbildungen 5.6 und 5.7 in der Mitte dargestellt. Die Aktivierungen von Aufgabe T1 sind in den Abbildungen mit schwarzen kleinen Pfeilen am unteren Rand der Diagramme markiert und die Aktivierungen von Aufgabe T2 durch graue kleine Pfeile. Eine weitere Aktivierung von T2 innerhalb von $3,25\,ms$ bereits nach $1,2\,ms$, anstatt wie erwartet erst nach $2,4\,ms$, benötigt weitere $70,73\,\mu J$, was zu einem erhöhten dynamischen Energieverbrauch von $212,19\,\mu J$ und einem Gesamtenergieverbrauch von $436,53\,\mu J$ führt. Die durchschnittliche dynamischen Verlustleistung von Aufgabe T2 steigt damit, wie in Abbildung 5.6 Mitte dargestellt, auf $65,29\,mW$ und die des Gesamtsystems auf $134,32\,mW$.

Wenn nun, wie in den Experimenten angenommen, die maximal verfügbare Energie bei $400\,\mu J$ liegt, müssen davon in jedem Fall $224,34\,\mu J$ für die kritische Aufgabe T1 garantiert sein. Damit darf der maximale Energieverbrauch der weniger kritischen Aufgabe T2 innerhalb des Messintervalls nicht über $175,66\,\mu J$ liegen. In Abbildung 5.7 Mitte kann man erkennen, wie die insgesamt verfügbare dynamische Energie durch die erhöhte Aktivität von T2 bereits nach $2,7\,ms$ verbraucht ist. Dies kann dazu führen, dass die kritische Aufgabe T1 nicht mehr korrekt ausgeführt werden kann.

Die Abbildungen 5.6 und 5.7 zeigen unten das Verhalten mit aktivierter automatischer Reaktion des im Rahmen dieser Arbeit entwickelten Laufzeitüberwachungsmechanismus. Der Mechanismus ist in dem dargestellten Experiment so programmiert, dass Kachel K2 deaktiviert wird, sobald ein nicht spezifiziertes Verhalten der ausgeführten weniger kritischen Aufgabe T2 erkannt wird, was nach

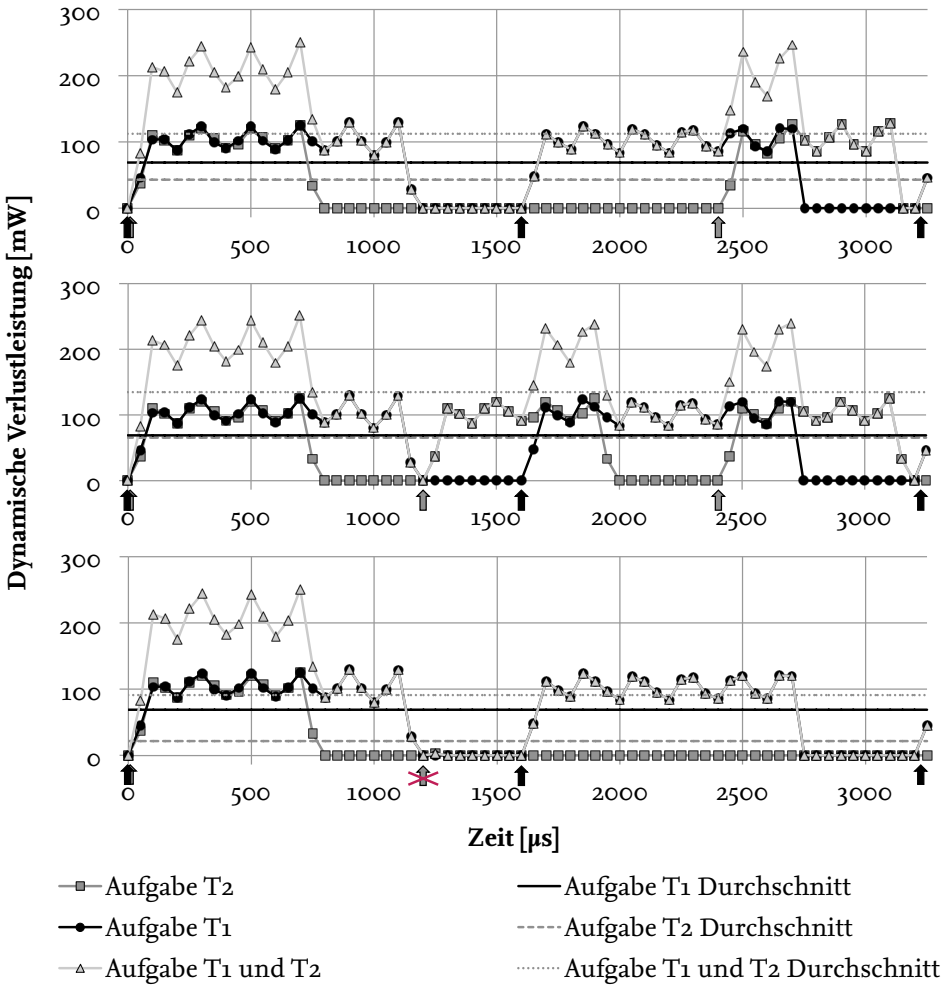

Abbildung 5.6.: Laufzeitüberwachung der dynamischen Verlustleistung

1,2 *ms* der Fall ist. Alternativ könnte Kachel K2 auch, wie in Abschnitt 4.2.10 beschrieben, angehalten werden bis weitere Aktivität nach 2,4 *ms* erwartet wird oder solange fortgeführt werden, solange T2 im aktuellen Messintervall nicht mehr Energie als 175,66 μJ verbraucht, um die kritische Aufgabe T1 nicht negativ zu beeinflussen. Damit dürfte Aufgabe T2 34,2 μJ zusätzlich, über die spezifizierte Energiemenge hinaus, verbrauchen.

Dies ist aber nur möglich, da der in Abschnitt 4.2.10 beschriebenen Mechanismus zur Separierung des Energieverbrauchs einzelner Anwendungen, wie die anderen Laufzeitüberwachungsmechanismen auch, dezentral in Hardware implementiert ist und so ohne Verzögerung einen erhöhten Energieverbrauch verhindern kann. Bei Lösungen, die auf der Interaktion mit einer zentralen Überwachungsinstanz beruhen, muss in der Analyse eine zusätzliche Energiereserve eingeplant werden, welche der maximalen Menge an Energie entspricht, welche eine weniger oder nicht kritische Aufgabe in der Zeit theoretisch verbrauchen kann, die von einem zentralen Überwachungsmechanismus maximal benötigt wird, einen erhöhten Energieverbrauch zu erkennen und weiteren Energieverbrauch zu verhindern. Da diese Zeit, wie in Abschnitt 3.4 erläutert, mit der Größe des Systems wächst, muss für größere System auch eine größere Energiereserve für weniger kritische oder nicht sicherheitsrelevante Aufgaben vorgehalten werden, was, wie bereits beschrieben, zu einer Überdi-

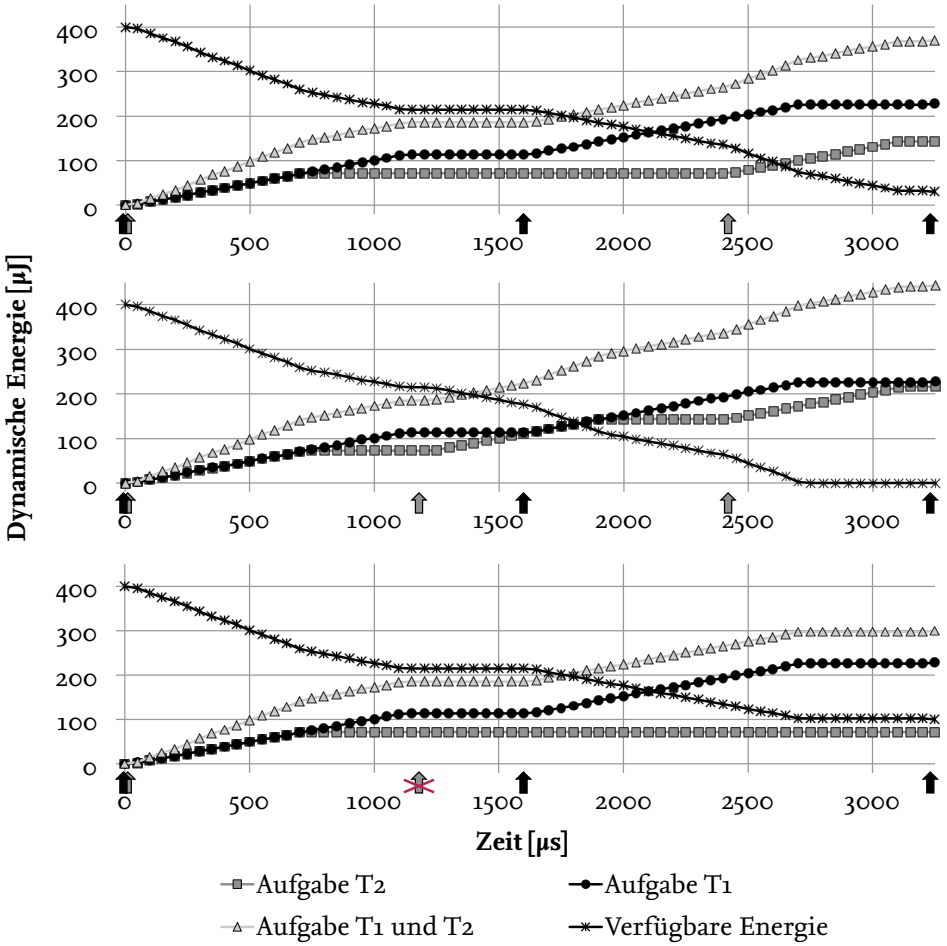

Abbildung 5.7.: Laufzeitüberwachung der dynamischen Energie

mensionierung des Systems führt. In dem obigen Beispiel ist eine maximale Energiereserve von $34,2\,\mu J$ möglich. Wenn in der maximalen Reaktionszeit eines zentralen Überwachungsmechanismus auf einer anderen Plattform mehr als diese Energiemenge von der weniger kritischen Aufgabe T2 im ungünstigsten Fall verbraucht werden könnte, wäre das obige Beispiel auf einer anderen Plattform mit einer zentralen Überwachungsinstanz nicht sicher ausführbar.

Die zusätzlich benötigten FPGA-Ressourcen für die Laufzeitüberwachung des Energieverbrauchs hängen von der Anzahl und der Genauigkeit der für die Energieabschätzung benötigten Ereignisse ab. Die Resultate von der Implementierung der in diesem Abschnitt verwendeten Laufzeitüberwachung des Energieverbrauchs mit einem lokalen Master, bis zu 16 lokalen Slaves und bis zu 16 verteilten Komponenten pro Kachel sind in Tabelle 5.1 bereits enthalten. Der zusätzliche Aufwand für die Laufzeitüberwachung des Energieverbrauchs ist eng mit der Größe der Adressübersetzungstabellen gekoppelt und beträgt in der dargestellten Konfiguration rund 2,4 %. Jeder weitere durch die Adressübersetzung repräsentierte Adressblock einer verteilten Komponente erhöht die Speicheranforderungen des Mechanismus um weitere acht Bits für die zusätzlich benötigte Energiegewichtung. Die Anzahl an Flipflops und LUTs pro Kachel bleibt, wie bei der Überwachung von gemeinsam verwendeten Komponenten, auch für große System konstant.

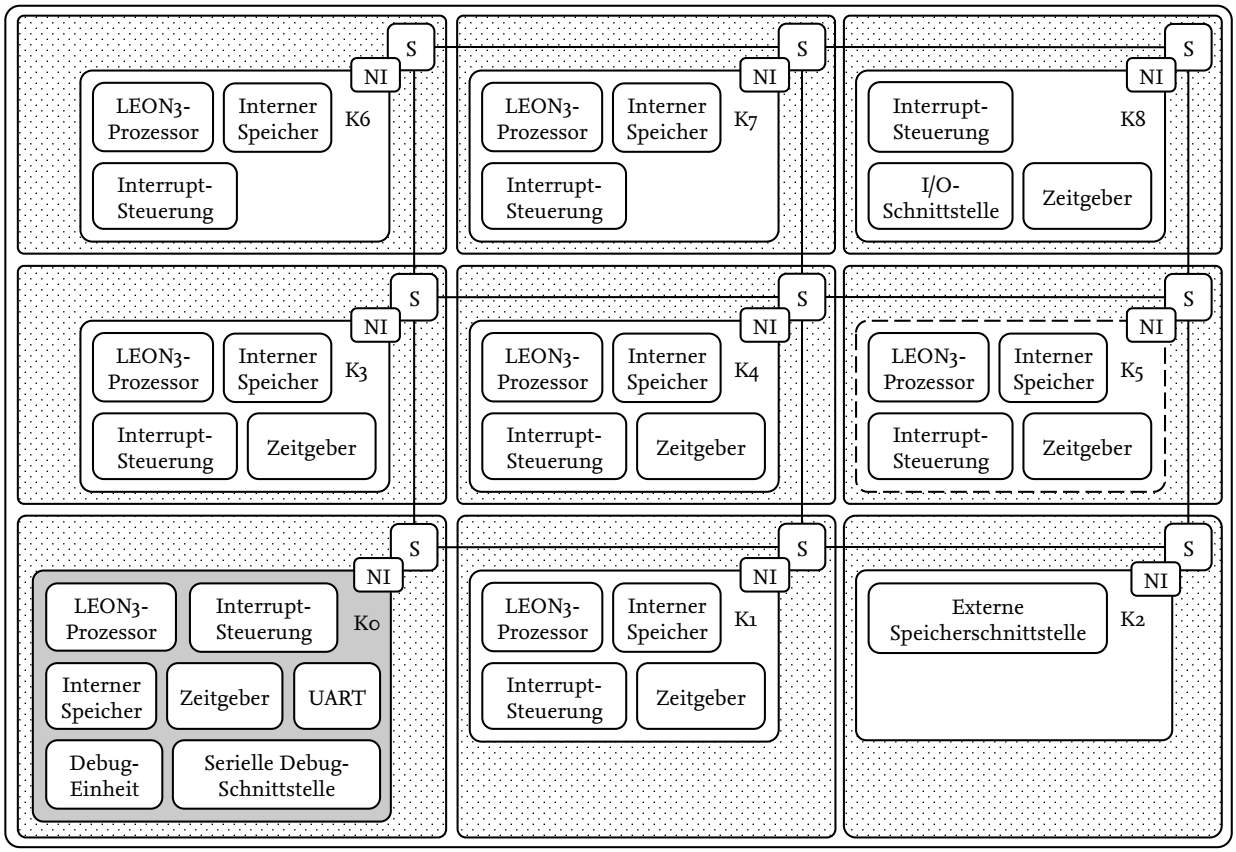

Abbildung 5.8.: IDAMC in größerer Konfiguration

5.3. Kommunikation

Um für die weiteren Experimente eine realistischere Konfiguration der IDAMC-Plattform zu generieren und um dabei die Skalierbarkeit der Plattform selbst zu evaluieren, wurde zusätzlich noch eine größere Konfiguration der IDAMC-Plattform auf dem HAPS-System implementiert. Die größere Konfiguration ist in Abbildung 5.8 dargestellt und besteht aus neun Switches mit jeweils einer angeschlossenen Kachel. Kachel 0 beinhaltet die gleichen Komponenten wie in der oben beschriebenen Minimalkonfiguration und ist auch hier wieder als Systemsteuerung konfiguriert. Kachel K1, K3, K4 und K5 sind identische Rechenkacheln und entsprechen weitestgehend den Rechenkacheln aus dem Minimalsystem, beinhalten hier jedoch zusätzlich einen kleinen lokalen Speicher von jeweils einem Kilobyte. Die Kacheln K6 und K7 sind ebenfalls als Rechenkacheln konfiguriert, verfügen jedoch über keinen lokalen Zeitgeber. Hierfür ist Kachel K8 als Peripheriekachel konfiguriert und beinhaltet neben einem Zeitgeber, eine I/O-Schnittstelle sowie eine lokale Interrupt-Steuerung. Der Zeitgeber in Kachel K8 kann von den Aufgaben in Kachel K6 und K7 gemeinsam verwendet werden. Kachel K2 ist als Speicherkachel konfiguriert und stellt eine Speicherschnittstelle zur Ansteuerung des externen DDR2-Speichers zur Verfügung.

5.3.1. Experimente

In den folgenden Experimenten führt Kachel Ko neben der Systemsteuerung eine weitere kritische Aufgabe To aus. Diese basiert wieder, wie in den obigen Experimenten ohne Einschränkung der

Allgemeingültigkeit, auf dem Dhrystone-Benchmark [90]. Aufgabe T0 wird von dem lokalen Zeitgeber in Kachel K0 alle 8,2 *ms* periodisch aktiviert und ihre Antwortzeit muss weniger als 7,6 *ms* betragen. Die Rechenkerne in den Kacheln K1, K3, K4, K6 und K7 führen weniger kritische oder auch nicht sicherheitsrelevante Aufgaben T1, T2, T3, T4 und T5 aus, welche periodisch mit der kritischen Aufgabe auf Kachel K0 über Interrupt-Anfragen und einen gemeinsam verwendeten Speicherbereich kommunizieren.

Die Aufgaben T1 bis T5 könnten zum Beispiel eine Vorverarbeitung von Sensordaten von Raddrehzahlsensoren oder von Feuchtigkeitssensoren darstellen, welche redundant und eventuell auch divers ausgeführt werden, um die Fehlertoleranz zu erhöhen. Ein weiteres mögliches Szenario sind Aufgaben, deren Ergebnisse nicht alle zwangsläufig für eine korrekte Ausführung der kritischen Aufgabe T0 benötigt werden, sondern zu einer Verbesserung der Genauigkeit der Ergebnisse dienen. Kachel K5 wird zunächst keine Aufgabe zugewiesen. Sie dient hier als Reservekachel und wird nur aktiviert, wenn in einer der anderen Rechenkacheln K1, K3 oder K4 ein permanenter Defekt angenommen wird.

Die weniger kritischen Aufgaben T1 bis T5 werden in den Experimenten in diesem Abschnitt periodisch aktiviert, entweder durch einen lokalen Zeitgeber, wie in Kachel K1, K3 und K4, oder, wie in Kachel K6 und K7, durch den gemeinsam verwendeten Zeitgeber der Peripheriekachel K8. Bei jeder Aktivierung schreiben die weniger kritischen Aufgaben neue Daten für die kritische Aufgabe T0 in den lokalen Speicher ihrer zugeordneten Kachel. Anschließend aktivieren die weniger kritischen Aufgaben jeweils einen Interrupt für den Rechenkern in Kachel K0, welcher über den Interrupt-Übersetzungsmechanismus in den jeweiligen Netzwerkschnittstellen als virtueller Rechenkern in die einzelnen Rechenkacheln eingeblendet wird. Nach Empfang einer Interrupt-Anfrage liest Aufgabe T0 die Nachricht aus dem lokalen Speicher der entsprechenden Rechenkachel für die weitere Verarbeitung. Aufgabe T0 kann über die Adressübersetzung in der Netzwerkschnittstelle der Kachel K0 auf die lokalen Speicher in den Rechenkacheln lesend zugreifen. In dem dargestellten Experiment werden die Nachrichten anschließend über die UART-Schnittstelle zur Auswertung an ein Terminal gesendet.

Die Aktivierungsperioden der weniger kritischen Aufgaben haben einen großen Einfluss auf die Antwortzeit der kritischen Aufgabe T0 auf Kachel K0. Bei jedem empfangenen Interrupt wird die kritische Aufgabe, wie in Abschnitt 3.2.2 erläutert, unterbrochen und somit verzögert. Abbildung 5.9 zeigt die Antwortzeit der kritischen Aufgabe T0 als Funktion der Anzahl eingehender Nachrichten von den anderen Kacheln innerhalb von 8,2 *ms*. Man kann erkennen, wie die Antwortzeit, abhängig von den Aktivierungsperioden der weniger kritischen Aufgaben auf den anderen Kacheln, proportional ansteigt und wie für mehr als 43 eingehende Interrupt-Anfragen innerhalb von 8,2 *ms* die Frist der kritischen Aufgabe T0 überschritten wird, womit diese fehlschlägt.

Die Frist der kritischen Aufgabe auf der Kachel K0 kann durch eine formale Analyse zur Entwicklungszeit, wie in Abschnitt 3.2 beschrieben, garantiert werden. Garantien aus der Analyse gelten jedoch nur, wenn sich alle Aufgaben, auch die weniger kritischen Aufgaben, zur Laufzeit genauso verhalten, wie bei der Analyse angenommen. Für weniger kritische Aufgaben kann dies ohne einen geeigneten Separierungsmechanismus oder einen erhöhten Zertifizierungsaufwand jedoch nicht als gegeben vorausgesetzt werden. Ein Fehler in einem der Zeitgeber, welche die weniger kritischen Aufgaben aktivieren, kann beispielsweise zu einer häufigeren Aktivierung und damit zu einem

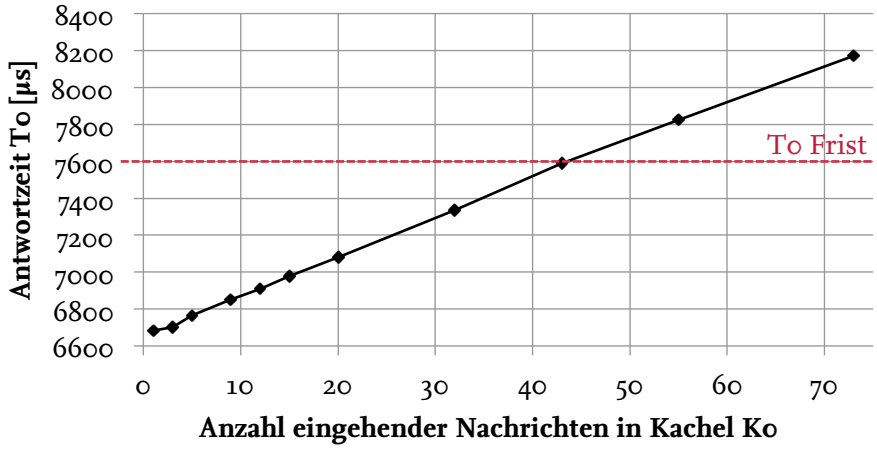

Abbildung 5.9.: Antwortzeit T0 ohne Laufzeitüberwachung

verkürzten Abstand zwischen aufeinanderfolgenden Nachrichten von der betroffenen Aufgabe an die kritische Aufgabe T0 führen.

Im Folgenden wird der Mechanismus zur Überwachung der Interrupt-Abstände eingesetzt, um den während der Analyse angenommenen minimalen Abstand zwischen aufeinanderfolgenden Interrupts von den weniger kritischen Aufgaben T1 bis T5 an die kritische Aufgabe T0 zur Laufzeit zu kontrollieren und gegebenenfalls zu erzwingen. Der erwartete minimale Abstand beträgt hierbei jeweils eine Millisekunde für alle eingehenden Nachrichten in Kachel K0. Abbildung 5.10 zeigt die Gesamtanzahl der Interrupt-Anfragen an Kachel K0 innerhalb einer Millisekunde und den individuellen Anteil der weniger kritischen Aufgaben auf den jeweiligen Rechenkacheln. Die ersten beiden Millisekunden werden für den Start des Systems benötigt und sind daher in der Abbildung nicht dargestellt.

In den Experimenten wird nach fünf Millisekunden ein Fehler in Kachel K4 injiziert und nach zehn Millisekunden in Kachel K8. Die Zeitpunkte der Fehlerinjektion sind in Abbildung 5.10 durch Pfeile gekennzeichnet. Bei der Fehlerinjektion wird jeweils die Zeitbasis des lokalen Zeitgebers in den beiden Kacheln manipuliert, um so den Abstand zwischen aufeinanderfolgenden Interrupts zu halbieren. Abbildung 5.10 oben zeigt das Experiment ohne und Abbildung 5.10 unten mit aktivierter Laufzeitüberwachung.

Ohne die Laufzeitüberwachung ist zu erkennen, wie die Anzahl der Nachrichten von den betroffenen Kacheln durch die Fehlerinjektionen nach fünf beziehungsweise nach zehn Millisekunden ansteigt. Da die Aufgaben T4 und T5 auf den Kacheln K6 und K7 den Zeitgeber in Kachel K8 gemeinsam verwenden, steigt sowohl die Anzahl der Nachrichten von Kachel K6 als auch von Kachel K7 nach einer Fehlerinjektion in Kachel K8.

Für die Auswahl geeigneter Reaktionen der dezentralen Laufzeitüberwachung wird für die Kacheln K1, K3 und K4 ein permanenter Defekt innerhalb der Kachel angenommen, wenn eine Abweichung der Interrupt-Abstände erkannt wird. Der Mechanismus ist daher so konfiguriert, dass die betroffene Kachel deaktiviert wird und die entsprechende Aufgabe auf die Reservekachel K5 migriert wird. In Abbildung 5.10 unten ist nach sechs Millisekunden zu erkennen, dass trotz Fehlerinjektion in Kachel K4 kein zusätzlicher Interrupt empfangen wurde, da dieser durch den Überwachungsmechanismus blockiert wurde. Die nächste Nachricht von der betroffenen Aufgabe T3 trifft dann erst

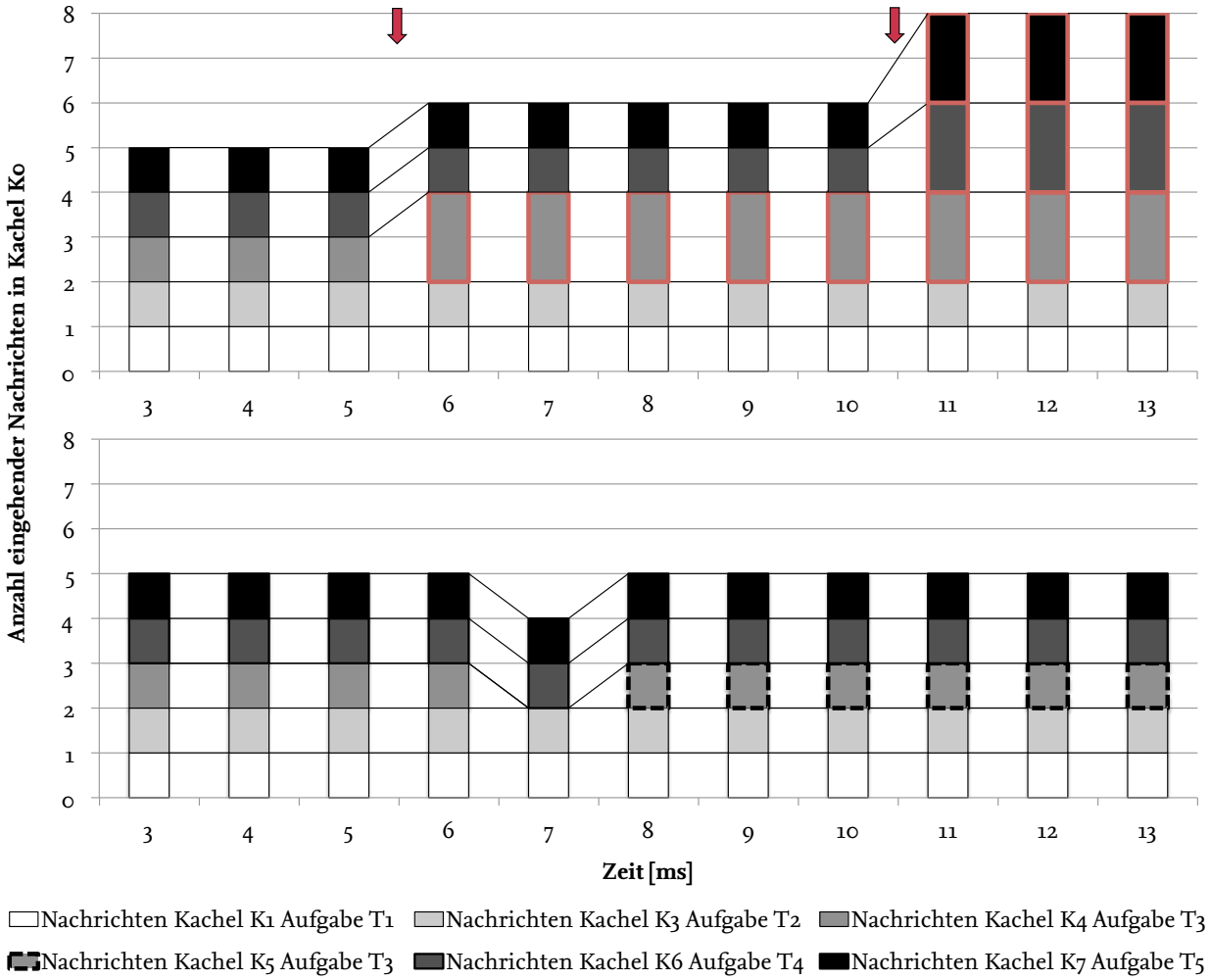

Abbildung 5.10.: Laufzeitüberwachung Kommunikation

nach weiteren zwei Millisekunden in Kachel Ko ein, da eine Millisekunde für die Migration von Aufgabe T3 von der defekten Kachel K4 auf die Reservekachel K5 benötigt wird.

Wenn der Abstand von aufeinanderfolgenden Interrupt-Anfragen von den Kacheln K6 und K7 zu kurz ist, wird von einem transienten Fehler innerhalb der Kachel K8 ausgegangen, da Kachel K6 und K7 den Zeitgeber in Kachel K8 gemeinsam verwenden. Die Peripheriekachel K8 wird daraufhin neu gestartet. Nach elf Millisekunden ist in Abbildung 5.10 unten zu erkennen, dass, anders als in Abbildung 5.10 oben, mit aktivierter Laufzeitüberwachung keine zusätzlichen Nachrichten von den Kacheln K6 und K7 empfangen werden. Der Neustart von Kachel K8 geschieht ausreichend schnell, so dass in Abbildung 5.10 unten keine Beeinträchtigung durch die Fehlerinjektion in Kachel K8 zu erkennen ist.

Da sämtliche zusätzliche Interrupt-Anfragen, welche nicht dem Minimalabstand des Ereignismodells entsprechen, in dem gezeigten Experiment bei aktivierter Laufzeitüberwachung blockiert werden, kann hier von einer ausreichenden Unabhängigkeit der kritischen Aufgabe To auf Kachel Ko von den weniger kritischen Aufgaben auf den anderen Kacheln ausgegangen werden.

Ressourcen	IDAMC	NoC	Switches	Kacheln	Interrupt-übersetzung	Interrupt-überwachung
#Flipflops	85626 (9 %)	41730	4203–5495	2384–5576	206	90
#LUTs	157135 (33 %)	68193	6564–9335	4573–12824	423	198
#BRAMs	182 (16 %)	33	3–5	3–79	1	1

Tabelle 5.5.: Syntheseergebnisse Drei-mal-Drei-System

5.3.2. Implementierung

Tabelle 5.5 zeigt die Resultate der Implementierung der größeren Drei-mal-Drei-Konfiguration der IDAMC-Plattform. Im Vergleich zu den Syntheseergebnissen der Zwei-mal-Zwei-Konfiguration aus Tabelle 5.1 kann man erkennen, dass das System wie erwartet nahezu linear mit der Anzahl der Kacheln und Switches wächst. Der Gesamtressourcenverbrauch des Systems mit neun Kacheln und neun Switches ist nicht exakt 9/4-mal so groß wie das kleinere System mit vier Kacheln und vier Switches, da sich zum einen die Größen der Kacheln durch die unterschiedlichen Ausprägungen unterscheiden und zum anderen der Ressourcenverbrauch der Switches von der jeweiligen Anzahl an Ports abhängt. Der mittlere Switch des Drei-mal-Drei-Systems mit fünf Ports benötigt beispielsweise deutlich mehr Ressourcen als ein Switch an den Ecken mit lediglich drei Ports, was den relativ großen Unterschied des Ressourcenverbrauchs bei den Switches in Tabelle 5.5 erklärt.

Der Interrupt-Übersetzungsmechanismus mit zugehöriger Laufzeitüberwachung, dessen Ressourcenverbrauch pro Kachel in Tabelle 5.5 in den Spalten sechs und sieben angegeben ist, unterstützt in der dargestellten Konfiguration für die Kacheln K0, K1, K3, K4, K5, K6 und K7 jeweils bis zu sechs und für Kacheln K8 bis zu sieben virtuelle Interrupt-Ziele mit jeweils 15 unterschiedlichen Interrupt-Nummern. Kachel K2 verfügt weder über eine lokale Interrupt-Steuerung, einen Interrupt-Übersetzungsmechanismus, noch über eine Laufzeitüberwachung, da Kachel K2 keine Komponenten beinhaltet, welche Interrupts auslösen könnten.

Für die übrigen Kacheln würde eine größere Anzahl an möglichen Interrupt-Zielen, wie bei der Laufzeitüberwachung von gemeinsam verwendeten Komponenten und des Energieverbrauchs, lediglich die Speicheranforderungen für die Übersetzungstabellen erhöhen, jedoch nicht den Verbrauch an Flipflop- oder LUT-Ressourcen der jeweiligen Kacheln, welcher auch für sehr große System konstant bleibt. Erhöhte Speicheranforderungen werden aber auch hier erst dann sichtbar, wenn die Größe der aktuell verwendeten BRAMs nicht mehr ausreicht.

Der in Abschnitt 4.2.9 vorgestellte Interrupt-Übersetzungsmechanismus erhöht zusammen mit der Interrupt-Überwachung den Logikverbrauch der IDAMC-Konfiguration auf dem FPGA um etwa drei Prozent. Insgesamt wächst der Logikverbrauch durch alle Übersetzungs- und Überwachungsmechanismen, inklusiver derer, welche in Tabelle 5.1 dargestellt sind, um etwa zehn Prozent.

Da der Logikmehrverbrauch pro Kachel auch für sehr große Systeme konstant auf einem niedrigen Niveau bleibt und der Speicherverbrauch mit der Anzahl an unterstützten Komponenten lediglich linear ansteigt, sind die vorgestellten Mechanismen gut für Vielkernplattformen geeignet. Der geringe Mehraufwand und die gute Skalierbarkeit beruhen vor allem auf der programmierbaren und dezentralen Implementierung der Mechanismen.

5.4. Diskussion

Die in der vorliegenden Arbeit vorgestellte IDAMC-Plattform ist die erste und bislang auch die einzige Plattform, welche die flexible, transparente, effiziente sowie sichere Integration von mehreren Anwendungen auf verteilten und gemeinsam verwendeten Ressourcen einer Vielkernplattform ermöglicht. Andere Plattformen, wie beispielsweise Intels SCC [47] oder Tileras TILE64 [14], erlauben zwar ebenfalls die Integration von mehreren Anwendungen, jedoch ohne die Möglichkeit, die Zuordnung von Anwendungen und Ressourcen zur Laufzeit transparent zu ändern oder unterschiedlich kritische Anwendungen sicher zu separieren.

Andere Systeme wiederum, wie zum Beispiel die Vielkernplattform der Arbeitsgruppe um Professor Herkersdorf [46], unterstützen zwar die Migration von Anwendungen auf andere Kacheln zur Laufzeit, erlauben jedoch ebenfalls nicht, verteilte Ressourcen ohne zusätzliche Software-Unterstützung transparent und sicher durch unterschiedlich kritische Anwendungen zu nutzen.

Einige Plattformen unterstützen die Integration von unterschiedlich kritischen Anwendungen, jedoch auf Kosten der Flexibilität und Effizienz. Aggarwal et al. präsentieren in [5] eine Plattform mit einer hohen Flexibilität zur Entwurfszeit sowie einer hohen Sicherheit zur Laufzeit durch eine komplett statische Trennung aller Systemkomponenten. Zur Laufzeit ist dadurch jedoch keinerlei Interaktion zwischen den Teilsystemen, keine gemeinsame und damit effiziente Nutzung von geteilten Komponenten, noch eine Migration von Anwendungen von stark ausgelasteten auf weniger stark ausgelasteten Teilsystemen möglich. Ähnlich verhält es sich mit der gruppierten Architektur aus dem parMERASA-Projekt [86], welche zwar über ein NoC zwischen den Teilsystemen eine begrenzte Interaktion ermöglicht, jedoch ebenfalls keine gemeinsame Verwendung von Komponenten, noch die Migration von Aufgaben zwischen den Teilsystemen für eine bessere Auslastung des Systems erlaubt.

Das ACROSS MPSoC [75] ähnelt der IDAMC-Plattform in einigen Punkten, wie beispielsweise den fehlereingrenzenden Teilsystemen, welche durch spezielle Netzwerkschnittstellen vom Rest des Systems separiert werden können. Bei dem ACROSS MPSoC sind die Teilsysteme jedoch in sich abgeschlossen und erlauben keine verteilte, transparente und flexible Implementierung von Anwendungen über mehrere Teilsysteme. Hierdurch können weder fehlerhafte Komponenten transparent ersetzt werden, noch zusätzliche Peripheriemodule und Rechenkerne lokal eingeblendet werden. Ein weiterer Unterschied ist das zeitgesteuerte NoC im Gegensatz zu dem ereignisgesteuerten NoC der IDAMC-Plattform. Durch das deterministische Verhalten ohne Konflikte zur Laufzeit erfordert das ACROSS MPSoC keine aufwendige Analyse zur Entwurfszeit, ist jedoch zur Laufzeit weniger effizient, da Ressourcen ungenutzt bleiben können, und auch weniger flexibel. Die Änderung der Zeitplanung, um beispielsweise auf Fehler oder auf andere Veränderungen zu reagieren, ist beim NoC des ACROSS MPSoC nur zu festgelegten Zeitpunkten möglich und betrifft dann auch immer das gesamte System, was den Mechanismus durch die fehlende Skalierbarkeit einer globalen Zeitplanung auf den Einsatz in eher kleinen oder mittleren Systemen begrenzt.

Bei allen bisher in diesem Abschnitt genannten Vielkernplattformen benötigen Anwendungen, anders als bei der IDAMC-Plattform, detaillierte Kenntnisse über die Plattform. Existierende Anwendungen müssen daher aufwendig portiert werden und Neuentwicklungen müssen explizit für die entsprechende Plattform entwickelt werden. Als Alternative, oder bei manchen der genannten Plattform auch als Ergänzung, bieten sich hier Virtualisierungslösungen an. Klassische Virtualisie-

rungslösungen, wie zum Beispiel NOVA [81] oder OKL4 [44], können die verfügbaren Ressourcen einer Plattform effizient verwalten und die gemeinsame Verwendung von unterschiedlich kritischen Anwendungen sicher voneinander separieren.

Die klassischen Virtualisierungslösungen benötigen jedoch für die vollständige Virtualisierung eine zusätzliche Hardware-Unterstützung, welche auf den meisten eingebetteten Systemen nicht vorhanden ist, oder erfordern im Falle der Paravirtualisierung die aufwendige Anpassung der Gastsysteme. Weiterhin kann ein zentraler Hypervisor auf einer Vielkernplattform zu einem Engpass und, bei einem Fehler im Hypervisor selbst, möglicherweise zu einen Ausfall des Gesamtsystems führen (engl. *Single Point of Failure*), wenn der Hypervisor bei sämtlichen Aktivitäten der Plattform involviert ist. Verteilte, speziell für Mehrkernplattform entwickelte Betriebssysteme, wie Barrelfish [13] oder RTEMS [65], sind besser für Vielkernplattformen geeignet, unterstützen jedoch keine heterogenen Systeme und das Ersetzen von fehlerhaften Komponenten zur Laufzeit. Weiterhin bleibt die Implementierung von hardwarespezifischen Funktionen, wie beispielsweise für die Kommunikation zwischen Rechenkernen, dem Nutzer überlassen. Daher würden Barrelfish und RTEMS unter anderem von dem sicheren und flexiblen Kommunikationsmechanismus, welcher in dieser Arbeit vorgestellt wird, profitieren können.

Die IDAMC-Plattform beinhaltet aber nicht nur einen Mechanismus für die virtuelle Kommunikation, sondern bietet eine komplette, hauptsächlich auf Hardware-Mechanismen basierende Virtualisierungslösung. Herkömmliche Einzel- und Mehrprozessoranwendungen laufen ohne Software-Anpassungen sicher voneinander separiert auf unterschiedlichen, möglicherweise auch heterogenen Kacheln. Weitere Rechenkerne, Peripheriemodule und Speicherschnittstellen können über die Adress- und Interrupt-Übersetzung lokal eingeblendet werden. Die Betriebssystemkerne können weiterhin getrennt von den Anwendungen im Kernelbereich ausgeführt werden, ohne ein Sicherheitsrisiko darzustellen. Die Virtualisierungsschicht wird auf der IDAMC-Plattform über separate Kacheln als unabhängige Partitionen, über programmierbare Hardware-Mechanismen in den Netzwerkschnittstellen sowie die zentrale Systemsteuerung realisiert. Die Systemsteuerung läuft parallel zu den Anwendungen und ist lediglich für die Konfiguration der einzelnen Übersetzungs- und Überwachungsmechanismen zuständig. Sie ist daher nicht an der eigentlichen Ausführen der einzelnen Anwendungen beteiligt, verzögert diese daher also, anders als Software-Virtualisierungslösungen, nicht. Weiterhin ist die benötige Menge an Code für die Systemsteuerung, also die *Trusted Computing Base*, selbst im Vergleich mit Mikrokerneln gering. Darüber hinaus ist die Systemsteuerung auf einer separaten Kachel zusätzlich räumlich von den Gastsystemen getrennt. Die IDAMC-Plattform schließt jedoch auch eine Verwendung einer zusätzlichen, klassischen Software-Virtualisierungslösung nicht aus.

Die räumliche Separierung der einzelnen Anwendungen geschieht auf der IDAMC-Plattform über die Zuweisung von unterschiedlichen Kacheln sowie die Adress- und Interrupt-Übersetzung in den Netzwerkschnittstellen. Die Adressübersetzung kann als weitere Speicherschutzeinheit für eine mehrstufige räumliche Separierung von Speicherzugriffen für Mehrkernplattformen, wie sie in [41] untersucht werden, angesehen werden. Hattendorf et al. untersuchen in [41] Konzepte für gemeinsame und individuelle Speicherschutzeinheiten beziehungsweise Speicherverwaltungseinheiten für die einzelnen Rechenkerne eines Mehrprozessorsystems. Das Fazit der Autoren läuft auf ein mehrstufiges Speicherschutzkonzept hinaus, aus einer gemeinsamen, vertrauenswürdigen MPU für die Separierung von Anwendungen auf unterschiedlichen Rechenkernen und zusätzlichen,

individuellen MPUs oder auch MMUs für die einzelnen Rechenkerne zur Separierung von einzelnen Aufgaben auf denselben Rechenkernen. Eine MPU wird gegenüber einer MMU aufgrund der besseren zeitlichen Vorhersagbarkeit als gemeinsame Einheit empfohlen. Das Verhalten einer MMU ist durch das eventuell nötige Nachladen von Übersetzungseinträgen schlechter zeitlich vorhersagbar, bietet jedoch eine flexible Möglichkeit zur Verwaltung eines virtuellen Adressraumes.

Der Adressübersetzungsmechanismus in den Netzwerkschnittstellen der IDAMC-Plattform kann mit einer vertrauenswürdigen, verteilten MMU verglichen werden, die zusätzlich zu den individuellen und optionalen MMUs der LEON3-Prozessoren in den Kacheln eingesetzt werden kann. Der Adressübersetzungsmechanismus in den Netzwerkschnittstellen besitzt die Vorteile einer MMU, um einen virtuellen Adressraum verwalten zu können, beinhaltet jedoch alle und nicht nur einen Teil der möglichen Einträge. Das schwierig vorhersagbare zeitliche Verhalten einer MMU entfällt daher, wodurch sich der in dieser Arbeit beschriebenen Adressübersetzungsmechanismus deterministisch verhält und damit auch für Echtzeitanwendungen eingesetzt werden kann.

Eine weitere Möglichkeit, Anwendungen auf einer gemeinsamen Vielkernplattform räumlich voneinander zu separieren, bietet die TILE64-Plattform [91]. Die fünf separaten, dedizierten Netzwerke ohne virtuelle Kanäle erlauben zwar keine logische Separierung von unterschiedlichen Anwendungen, wenn diese Pakete über dieselben Netzwerkverbindungen schicken, durch die *Multicore Hardwall* kann das System jedoch in einzelne Regionen für unterschiedliche Anwendungen aufgeteilt werden. Die *Multicore Hardwall* blockiert unerwünschten Verkehr an ausgehenden Verbindungen der Switches und benachrichtigt die lokale Anwendung auf dem betroffenen Knoten über einen Interrupt. Da unerwünschte Nachrichten jedoch nicht direkt an der Quelle unterbunden werden, wie in der vorliegenden Arbeit, kann ein zusätzlicher, unvorhergesehener Datenverkehr, welcher zu vermehrten Konfliktsituationen auch mit kritischen Paketen führen kann, nicht unterbunden werden. Weiterhin kann eine unbegrenzte Anzahl von Interrupts in dem betroffenen Knoten mit aktivierter *Hardwall* die möglicherweise kritische lokale Anwendung durch häufige Unterbrechungen so weit Verzögerung, dass sie ihre Frist verpasst.

Die zeitliche Separierung von unterschiedlich kritischen Anwendungen geschieht auf der IDAMC-Plattform über eine dynamische Laufzeitüberwachung. Die Überwachung und möglicherweise das Erzwingen eines bestimmten Verhaltens zur Laufzeit reduziert die Überabschätzung des erwarteten Verhaltens bei der Analyse und damit die Überdimensionierung des Systems. Weitere Vorteile in Bezug auf die effiziente Nutzung einer Vielkernplattform hat die dynamische Laufzeitüberwachung der IDAMC-Plattform gegenüber einer statischen, zur Entwicklungszeit festgelegten Ablaufplanung. Ein konfliktfreies Routing [75] und die konfliktfreie Verwendung von weiteren Ressourcen über festgelegte Zeitschlitze vermeidet zwar eine aufwendige Analyse, lässt jedoch Ressourcen ungenutzt und verhindert dynamische Reaktionen zur Laufzeit, beispielsweise zur Fehlerbehandlung oder zur Optimierung des Systems. Die Laufzeitüberwachungsmechanismen der IDAMC-Plattform ermöglichen die effiziente Ausnutzung der verfügbaren Ressourcen, vor allem für weniger kritische Anwendungen, denen ein abweichendes Verhalten zugestanden werden kann, solange alle kritischen Aufgaben sicher ausgeführt werden können.

Es existieren weitere, teilweise ähnliche Mechanismen für die Überwachung von Anwendungen zur Laufzeit. Durch eine zentrale Implementierung und das Fehlen von automatischen Reaktionen sind diese jedoch nicht geeignet, um unterschiedlich kritische Aufgaben auf einer gemeinsamen Plattform sicher und effizient voneinander zu separieren.

Ein ähnlicher wie der in der vorliegenden Arbeit vorgestellte Überwachungsmechanismus für eine Vielkernplattform wurde von Fiorin et al. in [36] vorgestellt. Der beschriebenen Mechanismus ist ebenfalls in den Netzwerkschnittstellen der einzelnen Kacheln implementiert und erlaubt die Überwachung von gemeinsam verwendeten Komponenten, des ausgehenden Datenverkehrs und weiterer Eigenschaften. Der Mechanismus ist flexibler als die Laufzeitüberwachung, welche in der vorliegenden Arbeit präsentiert wird, benötigt dafür aber so viele Ressourcen, dass er nicht für größere Systeme und für viele Verbindungen eingesetzt werden kann. Weiterhin verfügt der Überwachungsmechanismus von Fiorin et al. über keine automatischen Reaktionen, so dass im Falle einer erkannten Abweichung, eine Interaktion mit einer zentralen Instanz nötig ist. Die dadurch benötigte Reaktionszeit, welche darüber hinaus mit der Größe des Systems ansteigt, verhindert eine effiziente Nutzung der Plattform für unterschiedlich kritische Anwendungen. Der beschriebenen Überwachungsmechanismus ist also eher für die Überwachung einzelner Eigenschaften und nicht für die Separierung unterschiedlich kritischer Anwendungen geeignet.

Weitere Überwachungsmechanismen für Vielkernplattformen werden in [28] und [34] beschrieben. Beide Mechanismen erlauben die Überwachung einzelner Eigenschaften eines NoCs und die Vorverarbeitung der gesammelten Daten bereits in den Überwachungsinstanzen, um vor allem das Datenaufkommen zu reduzieren. Eine automatische Reaktion direkt in den Überwachungsinstanzen bietet jedoch keine der beschriebenen Mechanismen. Es wird stets die Kommunikation mit einer weitere Instanz benötigt, um eine Reaktion im Fehlerfall auszulösen zu können, was zu einer nicht vernachlässigbaren und mit der Systemgröße wachsenden Reaktionszeit führen würde.

Ein Mechanismus, welcher die Isolation von fehlerhaften Elementen einer Vielkernplattform ermöglicht, wird in [43] erläutert. Der Mechanismus basiert auf der gegenseitigen periodischen Überwachung von Elementen, was wiederum zu einer signifikanten Reaktionszeit führen kann. Weiterhin wird in der vorgestellten Lösung bei einer erkannten Abweichung stets das gesamte System neu gestartet, wodurch alle Anwendungen, auch unterschiedlich kritische Anwendungen, voneinander abhängig werden, da der Ausfall eines weniger kritischen Elements zu einem Neustart auch der kritischen Elemente führt.

Es existieren weitere Mechanismen, die speziell für die Überwachung des Interrupt-Aufkommens auf einer Plattform entworfen wurden. Hierbei kann zwischen Überwachungslösungen an der Interrupt-Quelle, am Interrupt-Ziel, Lösungen in Software oder auch in Hardware unterschieden werden. Eine Software-Implementierung einer Laufzeitüberwachung des Interrupt-Aufkommens am Ziel, wie in [37, 57, 33] dargestellt, ist für Vielkernplattformen und andere Systeme, bei denen Interrupts über ein geteiltes Kommunikationsmedium wie ein NoC gesendet werden, eher ungeeignet, da die übermäßig versendeten Interrupt-Pakete in diesem Fall immer noch das NoC belasten können. Weiterhin lassen sich Interrupt-Nummern damit nicht mehr von mehreren Quellen gemeinsam, aber unabhängig voneinander verwenden, da bei einer Überwachung am Ziel dann immer die Interrupts von allen Quellen deaktiviert werden würden. Hardware-Lösungen an dem Interrupt-Ziel wie in [40, 82] haben, neben dem teils erheblichen Ressourcenverbrauch, dieselben Nachteile wie Software-Lösungen.

Überwachungsmechanismen an dem Interrupt-Ziel, welche den Interrupt direkt an der Quelle deaktivieren würden, könnten das obige Problem lösen, die hierfür benötigte Reaktionszeit würde jedoch keine effiziente und skalierbare Realisierung für Vielkernplattformen zulassen.

Software-Lösungen an der Quelle der Interrupts, wie beispielsweise auch durch Virtualisierungslösungen realisierbar, bieten sich nur für Rechenkerne an. Peripheriemodule ohne die Möglichkeit ein Überwachungsprogramm auszuführen, können hiermit nicht überwacht werden. Eine Hardware-Lösung an der Quelle wird in [10, 67] präsentiert. Hier wird jedoch eine zentrale Einheit benötigt, um die einzelnen Überwachungsinstanzen zu steuern. Zum einen lassen sich Lösungen mit einer zentralen Einheit nicht für große Systeme skalieren und zum anderen können bei einem AMP-System die Zugriffe von unterschiedlich kritischen Aufgaben auf eine zentrale Einheit schwierig voneinander separiert werden.

Der letzte Punkt zeigt ein generelles Problem der Interrupt-Behandlung auf Vielkernprozessoren mit unterschiedlich kritischen Anwendungen, welches bisher nur ungenügend gelöst wurde. Interrupts werden auf Mehrkernplattformen, wie beispielsweise Intels SCC [50, 74], entweder durch eine zentrale Interrupt-Steuerung behandelt oder explizit an die Zielrechenkerne gesendet. Die erste der beiden Varianten ist nicht skalierbar für sehr große Vielkernplattformen mit vielen Interrupt-Quellen und -Zielen. Außerdem lassen sich Zugriffe von unterschiedlich kritischen Aufgaben auf eine zentrale Instanz schwierig ausreichend voneinander trennen. Das direkte Senden von Interrupts an das jeweilige Ziel verhindert dafür eine flexible und transparente Anpassung von Interrupt-Quellen und -Zielen. Das transparente Ersetzen von Interrupt-Zielen lässt sich gegebenenfalls noch über das Maskieren des alten Interrupt-Ziels und das Demaskieren des neuen Interrupt-Ziels erreichen. Das Ersetzen einer Interrupt-Quelle ist hiermit jedoch nicht möglich, da die Quelle stets an derselben Adresse erwartet wird, sofern Software-Änderungen hier ausgeschlossen werden. Die Virtualisierung der auf Interrupts basierenden Kommunikation für Vielkernplattformen, welche im Rahmen der vorliegenden Arbeit entwickelt wurde, bietet dagegen eine sichere, flexible und skalierbare Lösung für die genannten Probleme.

Ein weiterer Punkt, der in der Literatur, jedenfalls für sicherheitskritische Systeme und Systeme mit unterschiedlich kritischen Anwendungen, kaum Beachtung gefunden hat, ist die Separierung des Energieverbrauchs von unterschiedlich kritischen Anwendungen auf einer gemeinsamen Plattform. Da die Messung der Verlustleistung einzelner Anwendungen sowie der Energiedichte in einzelnen Chip-Regionen nicht möglich ist, wurden PMC und dedizierte Zähler für die Energieabschätzung bereits für viele verschiedene Plattformen und Anwendungsgebiete erfolgreich eingesetzt. Im Rahmen der vorliegenden Arbeit wurde der Ansatz der ereignisbasierten Energieabschätzung zum ersten Mal zur Separierung des Energieverbrauchs unterschiedlich kritischer Anwendungen eingesetzt. Die Methode zur Abrechnung von virtuellen Maschinen auf Servern abhängig vom Energieverbrauch [17] ähnelt der in dieser Arbeit vorgestellten Methode noch am meisten. Würde man die Berechnung des Energieverbrauchs der einzelnen virtuellen Maschinen mit einer oberen Grenze und einer automatischen Reaktion verbinden, hätte man eine entsprechende Software-Lösung zu der in dieser Arbeit vorgestellten hauptsächlich auf Hardware basierenden Lösung. Bei einer solchen Software-Lösung würde dann aber noch der Rechenmehraufwand besonders für Plattformen mit vielen Anwendungen dazukommen, wodurch eine solche Lösung eher für kleinere und mittlere Systeme geeignet wäre.

5.5. Zusammenfassung

In diesem Kapitel wurde die Funktionsweise der IDAMC-Plattform und insbesondere auch die Mechanismen zur Separierung unterschiedlich kritischer Anwendungen anhand von Experimenten erläutert. Hierbei lag der Schwerpunkt auf der Überwachung der Verwendung gemeinsamer Kom-

ponenten, wie zum Beispiel Speicherschnittstellen, der Kommunikation zwischen unterschiedlich kritischen Anwendungen über Kachelgrenzen hinweg und des Energieverbrauchs einzelner Anwendungen. Hierbei wurde auch die Charakterisierung von Ereignissen zur Modellierung des Energieverbrauchs der Plattform im Detail vorgestellt. Die beschriebenen Experimente zeigen die Beeinflussung von kritischen Aufgaben durch weniger kritische Aufgaben sowohl mit als auch ohne aktivierte Laufzeitüberwachung. Dabei wurde auch das Verhalten im Falle einer Abweichung des spezifizierten Verhaltens von weniger kritischen Aufgaben durch Fehlerinjektionen gezeigt. Weiterhin wurden Syntheseergebnisse der Implementierung von Beispielkonfigurationen der IDAMC-Plattform auf einem FPGA sowie andere Ansätze aus dem Umfeld dieser Arbeit ausführlich diskutiert.

6 Zusammenfassung

In der vorliegenden Arbeit wurde erläutert, warum Vielkernplattformen auch für eingebettete und sicherheitskritische Systeme attraktiv sind. Sollen mehrere sicherheitskritische oder auch unterschiedlich kritische Anwendungen auf einer gemeinsamen Vielkernplattform integriert werden, ist jedoch ein besonderer Augenmerk auf eine ausreichende Separierung zu legen. Eine ausreichende Separierung erlaubt es, die Anforderungen an weniger kritische Anwendungen gering zu halten, auch wenn diese gemeinsam mit hoch kritischen Anwendungen auf einer Plattform integriert werden. Auf diese Weise können die Gesamtkosten der Zertifizierung und einer eventuellen Neuzertifizierung nach der Aktualisierung einzelner Anwendungen reduziert werden, um so die Attraktivität von Vielkernplattformen für sicherheitskritische und unterschiedlich kritische Anwendungen zu steigern.

Die Separierung kann über eine strikte Trennung aller verfügbaren Ressourcen einer Vielkernplattform realisiert werden, was jedoch deren Vorteile stark einschränken würde. Die im Rahmen dieser Arbeit vorgestellte *Integrated Dependable Architecture for Many Cores (IDAMC)* bietet dagegen, neben einer klassischen räumlichen Trennung von Anwendungen über die exklusive Zuweisung von Kacheln, verteilten Komponenten sowie virtueller Kanäle des NoCs, eine neuartige zeitliche Separierung von unterschiedlich kritischen Anwendungen über eine dynamische Laufzeitüberwachung. Die Sicherstellung des erwarteten Verhaltens von weniger kritischen Anwendungen bei der Verwendung von gemeinsamen Ressourcen zusammen mit höher kritischen Anwendungen ermöglicht eine sichere und dennoch effiziente Ausnutzung der verfügbaren Ressourcen und verringert damit die Überdimensionierung des Systems.

Die vorgestellten Mechanismen zur Überwachung der einzelnen Anwendungen zur Laufzeit sind dezentral in den Netzwerkschnittstellen der einzelnen Kacheln in Hardware implementiert und erlauben so eine schnelle Reaktionszeit auf ein mögliches Fehlverhalten unabhängig von der Systemgröße und sind daher gut für Vielkernplattformen geeignet. Hierbei heben sich die Mechanismen vor allem gegenüber bestehenden Überwachungslösungen ab, welche eine Interaktion mit einer zentralen Überwachungs- oder Steuereinheit benötigen. Deren mangelnde Skalierbarkeit macht sie für den Einsatz auf Vielkernplattformen nicht empfehlenswert.

Neben der räumlichen und zeitlichen Separierung von unterschiedlich kritischen Anwendungen bietet die Plattform bislang als einzige auch die Möglichkeit, den Energieverbrauch, welcher von den einzelnen Anwendungen ausgeht, die Energiedichte in einzelnen Chip-Regionen sowie den individuellen Beitrag der einzelnen Anwendungen zur Energiedichte in gemeinsam verwendeten Regionen über eine ereignisbasierte Energieabschätzung sicher zu trennen, was über herkömmliche physikalische Messungen nicht möglich wäre.

Die Transparenz und Flexibilität der IDAMC-Plattform ermöglicht darüber hinaus, bestehende Anwendungen, ähnlich wie bei klassischen Virtualisierungslösungen, jedoch ohne spezielle Betriebssysteme oder Softwareanpassungen, leicht von Einzel- oder Mehrprozessorsystemen zu portieren. Auch die Entwicklung von neuen Anwendungen wird durch die skalierbaren Virtualisierungsme-

chanismen der IDAMC-Plattform erleichtert. Aufgrund der Implementierung in Hardware wird die Ausführung der Anwendungen hierbei jedoch nicht wie bei Software-Lösungen verzögert und das Risiko eines Ausfalls des Gesamtsystems durch einen Fehler in der Virtualisierungsschicht wird reduziert.

Die Zuordnung der Anwendungen zu möglicherweise auch heterogenen Kacheln und verteilten Ressourcen des Systems wird vor den Anwendungen verborgen und lässt sich gegebenenfalls zur Laufzeit ändern, um das System besser auszulasten oder die Fehlertoleranz einzelner Anwendungen zu erhöhen. Dies betrifft aber nicht nur die Zuordnung von Anwendungen, sondern auch die Kommunikation zwischen Anwendungen auf unterschiedlichen Kacheln und die Benachrichtigung durch verteilte Komponenten, was auf der IDAMC-Plattform über einen neuartigen, sicheren und skalierbaren, auf kachelübergreifenden Interrupts basierenden Kommunikationsmechanismus realisiert wird.

Die in der vorliegenden Arbeit vorgestellte parametrisierbare und vollständig synthetisierbare IDAMC-Plattform ist die erste und bislang auch die einzige Vielkernplattform, welche sowohl eine flexible, transparente und effiziente als auch eine sichere Integration von unterschiedlich kritischen Anwendungen auf verteilten und gemeinsam verwendeten Ressourcen erlaubt.

Der Schwerpunkt der vorliegenden Arbeit lag auf der sicheren Separierung von unterschiedlich kritischen Anwendungen auf einer Vielkernplattform ohne jedoch auf die sichere Ausführung von kritischen Anwendungen selbst genauer einzugehen. Es ist in der aktuellen Implementierung der IDAMC-Plattform daher nicht möglich, ohne zusätzlichen Aufwand hoch kritische Anwendungen ausreichend sicher zu integrieren. Hierfür müssten beispielsweise neben einer Reihe von weiteren Maßnahmen sowohl die Speicher des Systems als auch die eingesetzten Tabellen der eingeführten Übersetzungs- und Überwachungsmechanismen durch Fehlererkennungs- und Fehlerkorrekturmaßnahmen, wie zum Beispiel ECC, geschützt werden. Weiterhin müssten Fehler im NoC durch Maßnahmen wie redundante Routen oder Codierungen erkannt und korrigiert werden, oder auch Fehler in der Spannungsversorgung oder des Taktes, die das gesamte System betreffen würden, durch externe Maßnahmen abgesichert werden. Über solche bekannten Maßnahmen zur Risikoreduzierung von kritischen Anwendungen hinaus bieten Vielkernprozessoren, wie die IDAMC-Plattform, durch die vielen möglicherweise redundanten Ressourcen weitere Vorteile und Möglichkeiten, welche interessante Themen für zukünftige Forschungsarbeiten eröffnen.

A Programmierung

In diesem Abschnitt wird erläutert, wie auf die Hardware-Mechanismen zur flexiblen, transparenten und sicheren Ausführung von unterschiedlich kritischen Anwendungen auf der IDAMC-Plattform über die Software der Systemsteuerung zugegriffen werden kann. Die Systemsteuerung wird auf einer separaten Kachel des Systems parallel zu möglichen Betriebssysteminstanzen oder einzelnen Anwendungen auf den anderen Kacheln ausgeführt. Diese Betriebssysteminstanzen oder separaten Anwendungen benötigen keinerlei Kenntnis von der Existenz der Systemsteuerung. Sie werden ausgeführt wie auf einem herkömmlichen busbasierten Einzel- oder Multiprozessorsystem.

Bei Systemstart wird nur die Kachel der Systemsteuerung aktiviert. Alle anderen Kacheln bleiben zunächst im Reset-Zustand. Die Systemsteuerung konfiguriert die Hardware-Mechanismen in den Netzwerkschnittstellen über das NoC und aktiviert anschließend die entsprechenden Kacheln. Eine Umkonfiguration der Mechanismen zur Laufzeit ist ebenfalls möglich.

Um die Hardware-Mechanismen, welche in Abschnitt 4.2 im Detail erläutert werden, komfortabel über die Systemsteuerung programmieren zu können, wurden im Rahmen der vorliegenden Arbeit C-Funktionen entwickelt. Die Funktionen benötigen Kenntnisse über die tatsächlich vorhandenen Kacheln, deren jeweilige Konfiguration sowie die Topologie des NoCs. Die benötigten Informationen werden über eine Header-Datei bereitgestellt, welche automatisch aus der zentralen Konfigurationsdatei der IDAMC-Plattform generiert wird.

In den folgenden Unterabschnitten werden die wichtigsten C-Funktionen erläutert und ihre Funktionsweise an Beispielen verdeutlicht.

A.1. Adressübersetzung

Die Adressübersetzung blendet Adressblöcke aus anderen Kacheln in der Größe von entweder einem Megabyte für größere Speicheradressbereiche oder von 256 Bytes für kleinere I/O-Adressbereiche in den Adressbereich der Zielkachel ein. Die hierfür bereitgestellte Funktion *transl_addr*() übernimmt daher als Eingangsparameter die Identifikationsnummern der Quellkachel und der Zielkachel, sowie die Basisadressen des Adressblockes in der Quell- und der Zielkachel. Für größere Speicheradressbereiche sind die oberen zwölf Bits der Basisadresse, die Bits 31-20, signifikant und werden für die Adressierung der Übersetzungstabelle verwendet. Für kleinere I/O-Adressbereiche müssen die oberen zwölf Bits fest auf den Wert *AHBIO* gelegt werden, welcher in der zentralen Konfigurationsdatei festgelegt wird und über die generierte Header-Datei zugreifbar ist. Der Standardwert für *AHBIO* ist $0xFFF$. Für die Adressierung der Übersetzungstabelle für kleiner I/O-Adressbereiche werden dann die folgenden zwölf Bits, die Bits 19-8, verwendet. Die übrigen Bits werden sowohl bei Speicheradressbereichen als auch bei I/O-Adressbereichen für die Adressübersetzung nicht betrachtet.

Für die Basisadresse in der Zielkachel ist es wichtig zu beachten, dass sich diese in einem der Adressbereiche der zugehörigen Netzwerkschnittstelle befindet. Die Adressbereiche der Netzwerkschnittstelle werden, wie in Abschnitt 4.2.2 beschrieben, über Parameter der zentralen Konfigura-

tionsdatei festgelegt. Für die Kachel der Systemsteuerung gilt es weiterhin zu beachten, dass der unterste Adressblock von einem Megabyte in dem Speicheradressbereich für die Konfiguration der Netzwerkschnittstellen der übrigen Kacheln verwendet wird und so nicht für das Einblenden von anderen Speicherbereichen verwendet werden sollte.

Auf der IDAMC-Plattform ist sowohl die komplette Route von der Quell- zu der Zielkachel als auch der zu verwendende virtuelle Kanal in den Adressübersetzungstabellen spezifiziert. Es können hierfür zuvor statisch festgelegte Routen und virtuelle Kanäle aus der Header-Datei verwendet werden oder auch alternative Routen und Kanäle programmiert werden. Sowohl die virtuellen Kanäle als auch die Routen lassen sich auch zur Laufzeit ändern, um so beispielsweise auf Fehler im NoC reagieren zu können.

Weiterhin kann über die Funktion *transl_addr()* ein Zugriffsschutz für schreib- oder lesegeschützte Bereiche definiert werden. Die Funktion gibt bei erfolgreicher Ausführung die Adresse des geschriebenen Tabelleneintrages zurück und im Fehlerfall die Zahl Null.

Bibliotheksfunktion:

```
/*
 * Blendet einen Adressbereich aus einer anderen Kachel in der
 * Zielkachel ein.
 *
 * Parameter:
 * source_tile   Identifikationsnummer der Quellkachel
 * source_addr   Basisadresse in der Quellkachel
 * target_tile   Identifikationsnummer der Zielkachel
 * target_addr   Basisadresse in der Zielkachel
 * read          0=Leseschutz
 * write         0=Schreibschutz
 * route         Route von der Quell- zur Zielkachel
 * vc            Virtueller Kanal von der Quell- zur Zielkachel
 *
 * Rückgabewert:
 * Adresse des geschriebenen Tabelleneintrages, 0 im Fehlerfall
 */
int transl_addr(int source_tile, int source_addr, int target_tile,
                int target_addr, int read, int write,
                unsigned int route, unsigned int vc);
```

Beispiel Speicheradressbereich:

```
/*
 * Blendet den Speicheradressbereich 0x40300000-0x403FFFFF aus
 * Kachel 3 in den Adressbereich 0x10500000-0x105FFFFF in Kachel 1
 * schreibgeschützt ein.
 */
transl_addr(3,0x40300000,1,0x10500000,1,0,ROUTE[3][1],
            VIRTUAL_CHANNEL[3][1]);
```

Beispiel I/O-Adressbereich:

```
/*
 * Blendet den I/O-Adressbereich 0xFFF10000-0xFFF101FF aus Kachel 2 in
 * den Adressbereich 0xFFF00400-0xFFF005FF in Kachel 0 ein
 */
transl_addr(2,0xFFF10000,0,0xFFF00400,1,1,ROUTE[2][0],
            VIRTUAL_CHANNEL[2][0]);
transl_addr(2,0xFFF10100,0,0xFFF00500,1,1,ROUTE[2][0],
            VIRTUAL_CHANNEL[2][0]);
```

A.2. Interrupt-Übersetzung

Die Interrupt-Übersetzung erlaubt es, ähnlich wie die Adressübersetzung für verteilte Adressbereiche, Interrupt-Ziele aus anderen Kacheln lokal einzublenden. Hierfür müssen Interrupts von der lokalen Interrupt-Steuerung der Quellkachel an die Netzwerkschnittstelle geleitet werden. Anschließend können die Interrupts von der Interrupt-Übersetzung entsprechend der Übersetzungstabelle über das Netzwerk an die Zielkachel gesendet werden, wo sie wie andere lokale Interrupts aktiviert und von der lokalen Interrupt- Steuerung in der Zielkachel behandelt werden können.

Um die Interrupt-Übersetzungstabelle der Quellkachel zu programmieren und entsprechende Interrupts von der Interrupt-Steuerung an die Netzwerkschnittstelle zu leiten, kann die Systemsteuerung die C-Funktion *transl_irq()* verwenden. Die Funktion benötigt als Parameter die Quellkachel des Interrupts, die Identifikationsnummer des virtuellen Rechenkerns in der Quellkachel, welcher den physischen Rechenkern in der Zielkachel repräsentiert, die Route und den virtuellen Kanal, um das Interrupt-Paket über das NoC zu senden sowie ein Array, welches Interrupt-Nummern in der Quellkachel Interrupt-Nummern in der Zielkachel zuweist.

Bibliotheksfunktion:

```
/*
 * Blendet Interrupt-Ziele aus einer Kachel in einer anderen Kachel
 * lokal ein und konfiguriert die Interrupt-Steuerung der Quellkachel,
 * um entsprechende Interrupts an die Netzwerkschnittstelle zu leiten.
 *
 * Parameter:
 * tile           Identifikationsnummer der Quellkachel des Interrupts
 * core_id        Identifikationsnummer des virtuellen Zielrechenkerns
 * vchannel       Virtueller Kanal von der Quell- zur Zielkachel
 * route          Route von der Quell- zur Zielkachel
 * transl_irq_arr Zeiger auf ein Array, welches Interrupt-Nummern in
 *                der Quellkachel Interrupt-Nummern in der Zielkachel
 *                zuweist. Interrupt-Nummern, die nicht weitergeleitet
 *                werden sollen, müssen mit 0 initialisiert werden.
 */
void transl_irq(int tile, int core_id, int vchannel, int route,
                int* transl_irq_arr);
```

Beispiel:

```
/*
 * Blendet einen Rechenkern aus Kachel 5 als virtuellen Rechenkern mit
 * der Identifikationsnummer 4 in Kachel 3 ein und leitet den
 * Interrupt mit der Nummer 1 in der Quellkachel als Interrupt
 * mit der Nummer 2 in die Zielkachel weiter
 */
int transl_irq_arr[16]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

transl_irq_arr[1]=2;
transl_irq(3,4,VIRTUAL_CHANNEL[3][5],ROUTE[3][5],transl_irq_arr);
```

A.3. Laufzeitüberwachung

Um das Verhalten von Anwendungen zur Laufzeit zu überwachen und gegebenenfalls eine ausreichende Separierung, vor allem zwischen unterschiedlich kritischen Anwendungen, sicherzustellen, bieten die Netzwerkschnittstellen Überwachungsmechanismen an, welche in Abschnitt 4.2.10 beschrieben werden. Diese kann sich die Systemsteuerung in ihren lokalen Adressbereich ihrer eigenen Netzwerkschnittstelle einblenden, um die Mechanismen zu konfigurieren und gegebenenfalls überwachte Werte auszulesen. Die Auswertung der Daten kann entweder zentral durch die Systemsteuerung geschehen, wovon aus Gründen, welche in Abschnitt 3.4 erläutert werden, besonders für größere Systeme abgeraten wird, oder durch automatische, dezentrale Reaktionen in den Überwachungsmechanismen in den Netzwerkschnittstellen.

Diese Reaktionen umfassen den Neustart oder die Deaktivierung der betroffenen Kachel und das Erzwingen des erwarteten Verhaltens. Die Reaktionen können automatisch ausgelöst werden für den Fall, dass eine Abweichung vom erwarteten Verhalten erkannt wird. Weiterhin kann eine Nachricht an die Systemsteuerung für weitergehende Analysen und Reaktionen gesendet werden.

Die Konfiguration der einzelnen Mechanismen sowie das Einblenden der Konfigurationsbereiche in den lokalen Adressraum der Systemsteuerung kann ohne Kenntnis von Implementierungsdetails über die C-Funktionen, welche im Folgenden beschrieben werden, durchgeführt werden.

A.3.1. Gemeinsam verwendete Ressourcen

Die Überwachung von gemeinsam verwendeten Ressourcen geschieht über das Zählen von Zugriffen auf die entsprechenden Einträge in den Adressübersetzungstabellen in den Netzwerkschnittstellen der jeweiligen Kacheln. Um die Zugriffe auf gemeinsam verwendete Komponenten durch unterschiedlich kritische Anwendungen zu separieren, können Maximalwerte für Lese- und Schreibzugriffe innerhalb eines festgelegten Intervalls für einzelnen Tabelleneinträge aus der Analyse übernommen werden. Für die Überwachung und Separierung des Energieverbrauchs innerhalb der einzelnen Kacheln und von diesen ausgehend wird die Anzahl der Zugriffe auf die jeweiligen Tabelleneinträge mit einem zuvor ermittelten Gewicht multipliziert, welches der Energiemenge pro Zugriff entspricht. Die Energiegewichtungen werden über die Zeit, getrennt nach lokaler Energiedichte und Gesamtenergieverbrauch ausgehend von lokalen Anwendungen, aufsummiert. Die Einheit der Gewichtungen ist hier irrelevant solange sie mit der Einheit der Gesamtenergiebudgets übereinstimmt.

Für die Überwachung gemeinsam verwendeter Ressourcen werden zwei C-Funktionen bereitgestellt, eine für die Programmierung der einzelnen Tabelleneinträge und eine für Einstellung, welche die Laufzeitüberwachung unabhängig von den einzelnen Einträgen betreffen. Die letztgenannte Funktion aktiviert auch gleichzeitig den Überwachungsmechanismus.

Bibliotheksfunktionen:

```
/*
 * Spezifiziert die Energiegewichtungen und Maximalzählerstände
 * für einen Block der Adressübersetzung
 *
 * Parameter:
 * tile_nr         Kachelidentifikationsnummer
 * address         Zu überwachender Adressbereich
 * wenergy_weight  Energiemenge Schreibzugriff
 * renergy_weight  Energiemenge Lesezugriff
 * wcnt_max        Maximalanzahl Schreibzugriffe
 * rcnt_max        Maximalanzahl Lesezugriffe
 */
void monitor_addr(int tile_nr, int address, int wenergy_weight,
                  int renergy_weight, int wcnt_max, int rcnt_max);
```

```
/*
 * Stellt die allgemeinen Parameter der Laufzeitüberwachung
 * ein und aktiviert den Mechanismus
 *
 * Parameter:
 * tile_nr          Kachelidentifikationsnummer
 * emon_en          Aktiviert Energieüberwachung
 * emon_int_en      Aktiviert automatisches Zurücksetzen der
 *                  Energieakkumulatoren am Ende des Zeitintervalls
 * emon_msg_en      Aktiviert Nachricht an die Systemsteuerung bei
 *                  Überschreitung eines Energiebudgets
 * emon_pause_en    Verhindert weitere Aktivität der Kachel bis zum
 *                  Ende des aktuellen Zeitintervalls
 * emon_shtdwn_en   Deaktiviert Kachel bei Überschreitung eines der
 *                  Energiebudgets
 * emon_clr_read    Aktiviert das Rücksetzen der Energieakkumulatoren
 *                  nach einem Lesezugriff
 * emon_msg_irqnr   Interrupt-Nummer einer Nachricht an die System-
 *                  steuerung bei Überschreitung eines Energiebudgets
 * emon_int         Zeitintervall für Energieabschätzung
 * app_ebudget      Maximale Energiemenge, welche ausgehend durch die
 *                  lokale Anwendung verbraucht werden darf
 * tile_ebudget     Maximale lokale Energiedichte innerhalb der Kachel
```

```
 * cmon_en          Aktiviert Überwachung gemeinsamer Komponenten
 * cmon_int_en      Aktiviert automatisches Zurücksetzen der
 *                  Zählerstände am Ende des Zeitintervalls
 * cmon_msg_en      Aktiviert Nachricht an die Systemsteuerung bei
 *                  Überschreitung eines Maximalzählerstandes
 * cmon_pause_en    Verhindert weitere Zugriffe bis zum Ende des
 *                  aktuellen Zeitintervalls bei Überschreitung eines
 *                  Maximalzählerstandes
 * cmon_shtdwn_en   Deaktiviert Kachel bei Überschreitung einer der
 *                  Maximalzählerstände
 * cmon_clr_read    Aktiviert das Rücksetzen eines Zählerstandes nach
 *                  einem Lesezugriff
 * cmon_msg_irqnr   Interrupt-Nummer einer Nachricht an die
 *                  Systemsteuerung bei Überschreitung einer der
 *                  Maximalzählerstände
 * cmon_int         Zeitintervall für Zugriffsüberwachung
 */
void enable_addr_monitor(int tile_nr, int emon_en, int emon_int_en,
                         int emon_msg_en, int emon_pause_en,
                         int emon_shtdwn_en, int emon_clr_read,
                         int emon_msg_irqnr, int emon_int,
                         int app_ebudget, int tile_ebudget,
                         int cmon_en, int cmon_int_en,
                         int cmon_msg_en, int cmon_pause_en,
                         int cmon_shtdwn_en, int cmon_clr_read,
                         int cmon_msg_irqnr, int cmon_int);
```

Beispiel:

```
/*
 * Überwacht einen Adressbereich von einem Megabyte beginnend bei der
 * Adresse 0x40000000 in Kachel 1. Energiegewichtungen für Schreib-
 * und Lesezugriffe betragen 0x0A und 0x3C was hier 5 nJ
 * beziehungsweise 30 nJ entspricht. Maximalzählerstände betragen für
 * Schreibzugriffe 5000 und für Lesezugriffe 10000 innerhalb von
 * 1000000 Takten. In demselben Zeitraum dürfen von der Anwendung auf
 * der Kachel sowie innerhalb der Kachel 100 uJ verbraucht werden. Bei
 * Überschreitung eines Energiebudgets oder Maximalzählerstandes wird
 * die Kachel deaktiviert und eine Nachricht mit der Interrupt-Nummer
 * 6 beziehungsweise 7 wird an die Systemsteuerung gesendet.
 */
monitor_addr(1, 0x40000000, 0x0A, 0x3C, 0x01388, 0x02710);
enable_addr_monitor(1, 1, 1, 1, 0, 1, 1, 7, 0x000F4240, 0x00030D40,
                    0x00030D40, 1, 1, 1, 0, 1, 1, 6, 0x000F4240);
```

A.3.2. Interrupt-Abstände

Ein erhöhtes Interrupt-Aufkommen von anderen Kacheln kann das Zeitverhalten einer lokalen Aufgabe auf der Zielkachel negativ beeinflussen. Daher muss der Abstand zwischen aufeinanderfolgenden, ausgehenden Interrupts dem erwarteten und während der Analyse angenommenen Abstand entsprechen. Dies gilt insbesondere dann, wenn die Zielkachel eine kritische Aufgabe ausführt, aber auch, um vermehrte Konfliktsituationen innerhalb des NoCs durch eine erhöhte Anzahl an Interrupt-Paketen zu verhindern.

Um den Abstand zwischen aufeinanderfolgenden, ausgehenden Interrupts einer Kachel nach Interrupt-Nummern und virtuellen Zielprozessoren getrennt überwachen zu können, kann die C-Funktion *monitor_irq()* verwendet werden. Im Falle einer Unterschreitung des Minimalabstandes können automatische Reaktion ausgelöst werden. Die Funktion *enable_irq_monitor()* programmiert allgemeine Parameter des Überwachungsmechanismus und aktiviert diesen anschließend.

Bibliotheksfunktionen:

```
/*
 * Überwacht den Abstand zwischen aufeinanderfolgenden Interrupts
 * an virtuelle Rechenkerne in anderen Kacheln
 *
 * Parameter:
 * tile_nr          Quellkachel des zu überwachenden Interrupts
 * vcore_nr         Virtueller Zielkern des zu überwachenden Interrupts
 * irq_nr           Zu überwachende Interrupt-Nummer
 * min_distance     Geforderter Minimalabstand zwischen
 *                  aufeinanderfolgenden Interrupts in Anzahl Zyklen/256
 * delay            Reaktion: Verzögert den betroffenen Interrupt bis er
 *                  dem Minimalabstand entspricht
 * reset            Reaktion: Setzt die betroffene Kachel zurück
 * disable          Reaktion: Deaktiviert die betroffene Kachel
 * message          Reaktion: Sendet eine Nachricht an Systemsteuerung
 */
void monitor_irq(int tile_nr, int vcore_nr, int irq_nr,
                 int min_distance, int delay, int reset, int disable,
                 int message);
```

```
/*
 * Konfiguriert und Aktiviert die Interrupt-Überwachung
 *
 * Parameter:
 * tile_nr          Identifikationsnummer der zu überwachenden
 *                  Quellkachel
 * msg_irq          Gemeinsame Interrupt-Nummer für Nachrichten an die
 *                  Systemsteuerung
 */
void enable_irq_monitor(int tile_nr, int msg_irq);
```

Beispiel:

```
/*
 * Überwacht Interrupt-Nummer 2 in Kachel 1 an den virtuellen
 * Rechenkern 3. Der Minimalabstand beträgt 36864 Takte.
 * Im Falle einer Abweichung wird die betroffenen Kachel deaktiviert
 * und eine Nachricht mit der Interrupt-Nummer 7 an die
 * Systemsteuerung gesendet.
 */
monitor_irq(1, 2, 3, 0x000090, 0, 0, 1, 1);
enable_irq_monitor(1,7);
```

A.4. Kachelsteuerung

Da nur die Kachel mit der Systemsteuerung bei Systemstart aktiviert wird, müssen alle anderen Kacheln explizit durch die Systemsteuerung aktiviert werden. Weiterhin muss es der Systemsteuerung möglich sein, Kacheln zur Laufzeit wieder zu deaktivierten, beispielsweise im Fehlerfall. Hierfür steht die Funktion *enable_tile()* zur Verfügung.

Bibliotheksfunktion:

```
/**
 * Aktiviert, beziehungsweise deaktiviert, die genannte Kachel
 *
 * Parameter:
 * tile_number  Identifikationsnummer der zu
 *              aktivierenden/deaktivierenden Kachel
 * enable       1=Aktivieren, 0=Deaktivieren
 */
int enable_tile(int tile_number, int enable);
```

Beispiel:

```
/*
 * Aktiviert Kachel 1
 */
enable_tile(1,1);
```

Abkürzungsverzeichnis

AHB	Advanced High-Performance Bus
AMBA	Advanced Microcontroller Bus Architecture
AMP	Asymmetric Multiprocessing
APB	Advanced Peripheral Bus
ASIC	Application-Specific Integrated Circuit
ASIL	Automotive-Sicherheitsintegritätslevel
BRAM	Block Random Access Memory
CMOS	Complementary Metal-Oxide Semiconductor
COTS	Commercial Of-The-Shelf
DDR	Double Data Rate
DMA	Direct Memory Access
DMR	Dual Modular Redundancy
DRAM	Dynamic Random Access Memory
ECC	Error Correction Code
FIFO	First-In First-Out
Flit	Flow Control Digit
FPGA	Field Programmable Gate Array
FPU	Floating-Point Unit
GALS	Globally Asynchronous Locally Synchronous
GRLIB	Gaisler Research Library
GPIO	General-Purpose Input/Output
IDAMC	Integrated Dependable Architecture for Many Cores
IOMMU	Input/Output Memory Management Unit
IPC	Instructions Per Cycle
IRL	Interrupt Request Level

ISA	Instruction Set Architecture
ISR	Interrupt Service Routine
LUT	Lookup Table
MIMD	Multiple Instruction Multiple Data
MMU	Memory Management Unit
MPB	Message Passing Buffer
MPCI	Multiprocessor Communications Interface Layer
MPSoC	Multiprocessor System-on-Chip
MPU	Memory Protection Unit
NGMP	Next Generation Multipurpose Microprocessor
NI	Netzwerkschnittstelle
NoC	Network-on-Chip
PCIe	Peripheral Component Interconnect Express
Phit	Physical Digit
PMC	Performance Monitoring Counter
PROM	Programmable Read Only Memory
RTEMS	Real-Time Executive for Multiprocessor Systems
SCC	Single-Chip Cloud Computer
SDRAM	Synchronous Dynamic Random Access Memory
SEU	Single Event Upset
SIL	Sicherheitsintegritätslevel
SMP	Symmetric Multiprocessing
SMT	Simultaneous Multithreading
SoC	System-on-Chip
TDMA	Time Division Multiple Access
TLB	Translation Lookaside Buffer
TMR	Triple Modular Redundancy
TMU	Thermal Management Unit
VMM	Virtual Machine Monitor
WCRT	Worst-Case Response Time

Publikationen

[1] Motruk, Boris, Jonas Diemer, Philip Axer, Rainer Buchty und Mladen Berekovic: *Safe Virtual Interrupts Leveraging Distributed Shared Resources and Core-to-Core Communication on Many-Core Platforms.* In: *Dependable Computing (PRDC), 19th Pacific Rim International Symposium on*, Seiten 293–302. IEEE, 2013.

[2] Motruk, Boris, Jonas Diemer, Rainer Buchty und Mladen Berekovic: *Power Monitoring for Mixed-Criticality on a Many-Core Platform.* In: *Architecture of Computing Systems (ARCS)*, Seiten 13–24. Springer, 2013.

[3] Motruk, Boris, Jonas Diemer, Rainer Buchty, Rolf Ernst und Mladen Berekovic: *IDAMC: A Many-Core Platform with Run-Time Monitoring for Mixed-Criticality.* In: *High-Assurance Systems Engineering (HASE), 14th International Symposium on*, Seiten 24 –31. IEEE, 2012.

Literaturverzeichnis

[4] Aeroflex Gaisler: *GRLIB IP Core User's Manual*, 2011.

[5] Aggarwal, Nidhi, Parthasarathy Ranganathan, Norman P. Jouppi und James E. Smith: *Configurable Isolation: Building High Availability Systems with Commodity Multi-Core Processors.* ACM SIGARCH Computer Architecture News, 35(2):470–481, 2007.

[6] Andersson, J., J. Gaisler und R. Weigand: *Next Generation Multipurpose Microprocessor.* In: *Data Systems In Aerospace (DASIA)*, 2010.

[7] ARM: *AMBA Specification Rev. 2.0*, 1999.

[8] Atienza, David, Pablo G. Del Valle, Giacomo Paci, Francesco Poletti, Luca Benini, Giovanni De Micheli und Jose M. Mendias: *A Fast HW/SW FPGA-Based Thermal Emulation Framework for Multi-Processor System-on-Chip.* In: *Proceedings of the 43rd annual Design Automation Conference*, Seiten 618–623. ACM, 2006.

[9] Bachmann, Christian, Andreas Genser, Christian Steger, Reinhold Weiss und Josef Haid: *Automated Power Characterization for Run-Time Power Emulation of SoC Designs.* In: *Digital System Design: Architectures, Methods and Tools (DSD), 13th Euromicro Conference on*, Seiten 587–594. IEEE, 2010.

[10] Bak, S., E. Betti, R. Pellizzoni, M. Caccamo und L. Sha: *Real-Time Control of I/O COTS Peripherals for Embedded Systems.* In: *30th Real-Time Systems Symposium (RTSS)*, Seiten 193–203. IEEE, 2009.

[11] Baron, M.: *The Single-Chip Cloud Computer.* Microprocessor Report, 24(4), 2010.

[12] Baruah, Sanjoy, Haohan Li und Leen Stougie: *Towards the design of certifiable mixed-criticality systems.* In: *16th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Seiten 13–22. IEEE, 2010.

[13] Baumann, Andrew, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach und Akhilesh Singhania: *The Multikernel: A New OS Architecture for Scalable Multicore Systems.* In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, Seiten 29–44. ACM, 2009.

[14] Bell, S., B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown et al.: *TILE64 Processor: A 64-Core SoC with Mesh Interconnect.* In: *Solid-State Circuits Conference (ISSCC), Digest of Technical Papers*, Seiten 88–598. IEEE, 2008.

[15] Bellosa, F.: *The Case for Event-Driven Energy Accounting.* Technischer Bericht, University of Erlangen, Department of Computer Science, 2001.

[16] BELLOSA, FRANK: *The Benefits of Event-Driven Energy Accounting in Power-Sensitive Systems.* In: *Proceedings of the 9th workshop on ACM SIGOPS European workshop*, Seiten 37–42. ACM, 2000.

[17] BERTRAN, RAMON, YOLANDA BECERRA, DAVID CARRERA, VICENÇ BELTRAN, MARC GONZALEZ, XAVIER MARTORELL, JORDI TORRES und EDUARD AYGUADE: *Accurate Energy Accounting for Shared Virtualized Environments using PMC-based Power Modeling Techniques.* In: *Grid Computing (GRID), 11th IEEE/ACM International Conference on*, Seiten 1–8, 2010.

[18] BHATTACHARJEE, A., G. CONTRERAS und M. MARTONOSI: *Full-System Chip Multiprocessor Power Evaluations Using FPGA-Based Emulation.* In: *Low Power Electronics and Design (ISLPED), ACM/IEEE International Symposium on*, Seiten 335 –340, 2008.

[19] BHUNIA, SWARUP und SAIBAL MUKHOPADHYAY (Herausgeber): *Low-Power Variation-Tolerant Design in Nanometer Silicon.* Springer Verlag, 2010.

[20] BIRCHER, WILLIAM LLOYD und LIZY JOHN: *Predictive Power Management for Multi-Core Processors.* In: *Computer Architecture*, Seiten 243–255. Springer, 2012.

[21] BORKAR, S.: *Design Challenges of Technology Scaling.* Micro, IEEE, 19(4):23–29, 1999.

[22] BORKAR, SHEKHAR: *Thousand Core Chips - A Technology Perspective.* In: *Proceedings of the 44th annual Design Automation Conference*, Seiten 746–749. ACM, 2007.

[23] BUI, D., E. LEE, I. LIU, H. PATEL und J. REINEKE: *Temporal Isolation on Multiprocessing Architectures.* In: *48th Design Automation Conference (DAC)*, Seiten 274–279. IEEE, 2011.

[24] BURNS, ALAN und ROB DAVIS: *Mixed Criticality Systems-A Review.* Technischer Bericht, Department of Computer Science, University of York, Dec 2013.

[25] CHO, YOUNGJIN, YOUNGHYUN KIM, SANGYOUNG PARK und NAEHYUCK CHANG: *System-Level Power Estimation using an On-Chip Bus Performance Monitoring Unit.* In: *Computer-Aided Design (ICCAD), IEEE/ACM International Conference on*, Seiten 149 –154, 2008.

[26] CHUNG, S.W. und K. SKADRON: *Using On-Chip Event Counters For High-Resolution, Real-Time Temperature Measurement.* In: *Thermal and Thermomechanical Phenomena in Electronics Systems (ITHERM), The Tenth Intersociety Conference on*, Seiten 114–120. IEEE, 2006.

[27] CILKU, BEKIM und PETER PUSCHNER: *Towards Temporal and Spatial Isolation in Memory Hierarchies for Mixed-Criticality Systems with Hypervisors.* Proc. ReTiMiCS, RTCSA, Seiten 25–28, 2013.

[28] CIORDAŞ, CĂLIN, TWAN BASTEN, ANDREI RĂDULESCU, KEES GOOSSENS und JEF VAN MEERBERGEN: *An Event-based Monitoring Service for Networks on Chip.* ACM Transactions on Design Automation of Electronic Systems, 10(4):702–723, 2005.

[29] COBURN, JOEL, SRIVATHS RAVI und ANAND RAGHUNATHAN: *Power Emulation: A New Paradigm for Power Estimation.* In: *Proceedings of the 42nd annual Design Automation Conference*, Seiten 700 – 705. ACM, 2005.

[30] CONTRERAS, GILBERTO und MARGARET MARTONOSI: *Power Prediction for Intel XScale Processors Using Performance Monitoring Unit Events.* In: *Low Power Electronics and Design (ISLPED), International Symposium on*, Seiten 221–226. ACM, 2005.

[31] Del Valle, PG, D. Atienza, I. Magan, JG Flores, EA Perez, JM Mendias, L. Benini und G. De Micheli: *A Complete Multi-Processor System-on-Chip FPGA-Based Emulation Framework.* In: *Very Large Scale Integration, IFIP International Conference on*, Seiten 140–145. IEEE, 2006.

[32] Diemer, Jonas und Rolf Ernst: *Back Suction: Service Guarantees for Latency-Sensitive On-Chip Networks.* In: *Networks-on-Chip (NOCS), Fourth ACM/IEEE International Symposium on*, Seiten 155–162, 2010.

[33] Facchinetti, T., G. Buttazzo, M. Marinoni und G. Guidi: *Non-Preemptive Interrupt Scheduling for Safe Reuse of Legacy Drivers in Real-Time Systems.* In: *Real-Time Systems (ECRTS), 17th Euromicro Conference on*, Seiten 98 – 105. IEEE, 2005.

[34] Faruque, A., M. Abdullah, T. Ebi und J. Henkel: *ROAdNoC: Runtime Observability for an Adaptive Network on Chip Architecture.* In: *Computer-Aided Design (ICCAD), International Conference on*, Seiten 543–548. IEEE, 2008.

[35] Fernández, Mikel, Roberto Gioiosa, Eduardo Quiñones, Luca Fossati, Marco Zulianello und Francisco J Cazorla: *Assessing the Suitability of the NGMP Multi-core Processor in the Space Domain.* In: *Proceedings of the tenth ACM international conference on Embedded software*, Seiten 175–184. ACM, 2012.

[36] Fiorin, L., G. Palermo und C. Silvano: *A Configurable Monitoring Infrastructure for NoC-Based Architectures.* Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, 2013.

[37] Foyo, Luis E. Leyva-del, Pedro Mejia-Alvarez und Dionisio de Niz: *Integrated Task and Interrupt Management for Real-Time Systems.* ACM Transactions on Embedded Computing Systems (TECS), 11(2):32, 2012.

[38] Gaisler, Jiri und Sandi Habinc: *GRLIB IP Library User's Manual.* Aeroflex Gaisler, 2010.

[39] Genser, A., C. Bachmann, J. Haid, C. Steger und R. Weiss: *An Emulation-Based Real-Time Power Profiling Unit for Embedded Software.* In: *Systems, Architectures, Modeling, and Simulation (SAMOS), International Symposium on*, Seiten 67 –73. IEEE, 2009.

[40] Haid, J., G. Kaefer, C. Steger und R. Weiss: *A Co-Processor for Real-Time Energy Estimation of System-On-a-Chip.* In: *Circuits and Systems (MWSCAS), 45th Midwest Symposium on.* IEEE, 2002.

[41] Hattendorf, Anton, Andreas Raabe und Alois Knoll: *Shared Memory Protection for Spatial Separation in Multicore Architectures.* In: *Industrial Embedded Systems (SIES), 7th International Symposium on*, Seiten 299–302. IEEE, 2012.

[42] Hebert, N., G.M. Almeida, P. Benoit, G. Sassatelli und L. Torres: *A cost-effective solution to increase system reliability and maintain global performance under unreliable silicon in MPSoC.* In: *International Conference on Reconfigurable Computing and FPGAs*, Seiten 346–351. IEEE, 2010.

[43] Hebert, N., P. Benoit, G. Sassatelli und L. Torres: *D-Scale: A scalable system-level dependable method for MPSoCs.* In: *19th Asian Test Symposium*, Seiten 198–205. IEEE, 2010.

[44] HEISER, GERNOT und BEN LESLIE: *The OKL4 Microvisor: Convergence Point of Microkernels and Hypervisors.* In: *Proceedings of the first ACM asia-pacific workshop on Workshop on systems*, Seiten 19–24. ACM, 2010.

[45] HENIA, R., A. HAMANN, M. JERSAK, R. RACU, K. RICHTER und R. ERNST: *System level performance analysis - the SymTA/S approach.* Computers and Digital Techniques, IEEE, 152(2):148 – 166, 2005.

[46] HERKERSDORF, ANDREAS, HANS-ULRICH MICHEL, HOLM RAUCHFUSS und THOMAS WILD: *Multi-core Enablement for Automotive Cyber Physical Systems.* it-Information Technology, 54(6):280–287, 2012.

[47] HOWARD, J., S. DIGHE, S.R. VANGAL, G. RUHL, N. BORKAR, S. JAIN, V. ERRAGUNTLA, M. KONOW, M. RIEPEN, M. GRIES et al.: *A 48-Core IA-32 Processor in 45 nm CMOS Using On-Die Message-Passing and DVFS for Performance and Power Scaling.* Solid-State Circuits, IEEE Journal of, 46(1):173–183, 2011.

[48] HOYME, K. und K. DRISCOLL: *SAFEbus [for avionics].* Aerospace and Electronic Systems Magazine, IEEE, 8(3):34 –39, 1993.

[49] HUANG, WEI, MIRCEA R. STANT, KARTHIK SANKARANARAYANAN, ROBERT J. RIBANDO und KEVIN SKADRON: *Many-Core Design from a Thermal Perspective.* In: *45th Design Automation Conference (DAC)*, Seiten 746–749. IEEE, 2008.

[50] INTEL LABS: *The SccKit 1.4.0 User's Guide*, 2011.

[51] INTERNATIONAL ELECTROTECHNICAL COMMISSION: *61508 - Functional safety of electrical/electronic/-programmable electronic safety-related systems*, 2010.

[52] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION: *26262 - Road vehicles–Functional safety*, 2011.

[53] ISCI, C. und M. MARTONOSI: *Runtime Power Monitoring in High-End Processors: Methodology and Empirical Data.* In: *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, Seite 93. IEEE, 2003.

[54] KINSY, M.A., M. PELLAUER und S. DEVADAS: *Heracles: Fully Synthesizable Parameterized MIPS-Based Multicore System.* In: *Field Programmable Logic and Applications (FPL), International Conference on.* IEEE, 2011.

[55] KOTABA, ONDREJ, JAN NOWOTSCH, MICHAEL PAULITSCH, STEFAN M PETTERS und HENRIK THEILING: *Multicore In Real-Time Systems–Temporal Isolation Challenges Due To Shared Resources.* In: *Proceedings of Workshop on Industry-Driven Approaches for Cost-effective Certification of Safety-Critical, Mixed-Criticality Systems (WICERT)*, 2013.

[56] KRANICH, TIM und MLADEN BEREKOVIC: *NoC switch with credit based guaranteed service support qualified for GALS systems.* In: *Digital System Design: Architectures, Methods and Tools (DSD), 13th Euromicro Conference on*, Seiten 53–59. IEEE, 2010.

[57] Lemerre, Matthieu, Emmanuel Ohayon, Damien Chabrol, Mathieu Jan und M-B Jacques: *Method and tools for mixed-criticality real-time applications within PharOS.* In: *Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 14th International Symposium on*, Seiten 41–48. IEEE, 2011.

[58] Li, Tao und Lizy Kurian John: *Run-time Modeling and Estimation of Operating System Power Consumption.* In: *Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, Seiten 160–171. ACM, 2003.

[59] Lin, Tsung-Han, Y. Kinebuchi, A. Courbot, H. Shimada, T. Morita, H. Mitake, Chen-Yi Lee und T. Nakajima: *Hardware-Assisted Reliability Enhancement for Embedded Multi-core Virtualization Design.* In: *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 14th International Symposium on*, Seiten 241 –249, 2011.

[60] Malone, Michael: *OPERA RHBD multi-core.* In: *Military/Aerospace Programmable Logic Device Workshop (MAPLD)*, 2009.

[61] Martens, Arthur: *Implementierung von virtuellen Inter-Tile-Interrupts für eine LEON3-basierte Many-Core-Plattform.* Studienarbeit, TU Braunschweig, 2011.

[62] Merkel, Andreas und Frank Bellosa: *Balancing Power Consumption in Multiprocessor Systems.* In: *ACM SIGOPS Operating Systems Review*, Seiten 403–414. ACM, 2006.

[63] Mollison, Malcolm S, Jeremy P Erickson, James H Anderson, Sanjoy K Baruah und John A Scoredos: *Mixed-Criticality Real-Time Scheduling for Multicore Systems.* In: *Computer and Information Technology (CIT), 10th International Conference on*, Seiten 1864–1871. IEEE, 2010.

[64] Neukirchner, M., T. Michaels, P. Axer, S. Quinton und R. Ernst: *Monitoring Arbitrary Activation Patterns in Real-Time Systems.* In: *33rd Real-Time Systems Symposium (RTSS)*, Seiten 293–302. IEEE, 2012.

[65] On-Line Applications Research Corporation: *RTEMS C User's Guide*, 4.10.2 Auflage, 2011.

[66] Peddersen, Jorgen und Sri Parameswaran: *Low-Impact Processor for Dynamic Runtime Power Management.* IEEE Design & Test of Computers, 25(1):52 –62, 2008.

[67] Pellizzoni, R., E. Betti, S. Bak, Gang Yao, J. Criswell, M. Caccamo und R. Kegley: *A Predictable Execution Model for COTS-Based Embedded Systems.* In: *17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Seiten 269 –279, 2011.

[68] Pham, D.C., T. Aipperspach, D. Boerstler, M. Bolliger, R. Chaudhry, D. Cox, P. Harvey, P.M. Harvey, H.P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Pham, J. Pille, S. Posluszny, M. Riley, D.L. Stasiak, M. Suzuoki, O. Takahashi, J. Warnock, S. Weitzel, D. Wendel und K. Yazawa: *Overview of the Architecture, Circuit Design, and Physical Implementation of a First-Generation Cell Processor.* Solid-State Circuits, IEEE Journal of, 41(1):179 –196, 2006.

[69] Racu, Razvan, Arne Hamann, Rolf Ernst, Bren Mochocki und Xiaobo Sharon Hu: *Methods for Power Optimization in Distributed Embedded Systems with Real-Time Requirements.* In: *Proceedings*

of the 2006 international conference on Compilers, architecture and synthesis for embedded systems, Seiten 379–388. ACM, 2006.

[70] RAUCHFUSS, HOLM, THOMAS WILD und ANDREAS HERKERSDORF: *A Network Interface Card Architecture for I/O Virtualization in Embedded Systems.* In: *Proceedings of the 2nd conference on I/O virtualization*, 2010.

[71] RAUCHFUSS, HOLM, THOMAS WILD und ANDREAS HERKERSDORF: *Enhanced Reliability in Tiled Manycore Architectures through Transparent Task Relocation.* In: *ARCS Workshops (ARCS)*, Seiten 1 –6. IEEE, 2012.

[72] REGEHR, JOHN und USIT DUONGSAA: *Preventing Interrupt Overload.* In: *ACM SIGPLAN Notices*, Seiten 50–58. ACM, 2005.

[73] ROTEM, E., A. NAVEH, D. RAJWAN, A. ANANTHAKRISHNAN und E. WEISSMANN: *Power-Management Architecture of the Intel Microarchitecture Code-Named Sandy Bridge.* Micro, IEEE, 32(2):20–27, 2012.

[74] ROTTA, R.: *On Efficient Message Passing on the Intel SCC.* In: *MARC Symposium*, Seiten 53–58, 2011.

[75] SALLOUM, CHRISTIAN EL, MARTIN ELSHUBER, OLIVER HOFTBERGER, HARIS ISAKOVIC und ARMIN WASICEK: *The ACROSS MPSoC–A New Generation of Multi-core Processors Designed for Safety-Critical Embedded Systems.* In: *Digital System Design (DSD), 15th Euromicro Conference on*, Seiten 105–113. IEEE, 2012.

[76] SCHELER, F., W. HOFER, B. OECHSLEIN, R. PFISTER, W. SCHRÖDER-PREIKSCHAT und D. LOHMANN: *Parallel, Hardware-Supported Interrupt Handling in an Event-Triggered Real-Time Operating System.* In: *Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*, Seiten 167–174. ACM, 2009.

[77] SCHLIECKER, S.: *Performance Analysis of Multiprocessor Real-Time Systems with Shared Resources.* Doktorarbeit, Technische Universität Braunschweig, Germany, 2011.

[78] SCHLIECKER, SIMON und ROLF ERNST: *Real-Time Performance Analysis of Multiprocessor Systems with Shared Memory.* Transactions on Embedded Computing Systems (TECS), 10(2):22, 2010.

[79] SKADRON, K., M.R. STAN, K. SANKARANARAYANAN, W. HUANG, S. VELUSAMY und D. TARJAN: *Temperature-Aware Microarchitecture: Modeling and Implementation.* ACM Transactions on Architecture and Code Optimization (TACO), 1(1):94–125, 2004.

[80] SNOWDON, DAVID C., STEFAN M. PETTERS und GERNOT HEISER: *Accurate On-line Prediction of Processor and Memory Energy Usage Under Voltage Scaling.* In: *Proceedings of the 7th international conference on Embedded software*, Seiten 84–93. ACM, 2007.

[81] STEINBERG, UDO und BERNHARD KAUER: *NOVA: A Microhypervisor-Based Secure Virtualization Architecture.* In: *Proceedings of the 5th European conference on Computer systems*, Seiten 209–222. ACM, 2010.

[82] STRNADEL, J.: *Monitoring-Driven HW/SW Interrupt Overload Prevention for Embedded Real-Time Systems.* In: *Design and Diagnostics of Electronic Circuits Systems (DDECS), 15th International Symposium on*, Seiten 121 –126. IEEE, 2012.

[83] Synopsys: *HAPS-62 Hardware Reference Manual*, 2010.

[84] Takamaeda, S., S. Sato, T. Miyoshi und K. Kise: *Smart Core System for Dependable Many-Core Processor with Multifunction Routers.* In: *First International Conference on Networking and Computing*, Seiten 133–139. IEEE, 2010.

[85] Tumeo, A., M. Branca, L. Camerini, M. Monchiero, G. Palermo, F. Ferrandi und D. Sciuto: *An Interrupt Controller for FPGA-based Multiprocessors.* In: *Embedded Computer Systems: Architectures, Modeling and Simulation (IC-SAMOS), International Conference on*, Seiten 82–87. IEEE, 2007.

[86] Ungerer, T, C Bradatsch, M Gerdes, F Kluge, R Jahr, J Mische, J Fernandes, PG Zaykov, Z Petrov, B Boddeker et al.: *parMERASA–Multi-core Execution of Parallelised Hard Real-Time Applications Supporting Analysability.* In: *Digital System Design (DSD), Euromicro Conference on*, Seiten 363–370. IEEE, 2013.

[87] Vajda, András: *Programming many-core chips.* Springer, 2011.

[88] Vermeulen, B. und K. Goossens: *A Network-on-Chip Monitoring Infrastructure for Communication-centric Debug of Embedded Multi-Processor SoCs.* In: *VLSI Design, Automation and Test (VLSI-DAT), International Symposium on*, Seiten 183–186. IEEE, 2009.

[89] Villalpando, Carlos, David Rennels, Raphael Some und Manuel Cabanas-Holmen: *Reliable Multicore Processors for NASA Space Missions.* In: *Aerospace Conference*, Seiten 1–12. IEEE, 2011.

[90] Weicker, Reinhold P: *Dhrystone: A Synthetic Systems Programming Benchmark.* Communications of the ACM, 27(10):1013–1030, 1984.

[91] Wentzlaff, D., P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.C. Miao, J.F. Brown und A. Agarwal: *On-chip Interconnection Architecture of the Tile Processor.* Micro, IEEE, 27(5):15–31, 2007.

[92] Windsor, James und K. Hjortnaes: *Time and Space Partitioning in Spacecraft Avionics.* In: *Space Mission Challenges for Information Technology (SMC-IT), Third International Conference on*, Seiten 13–20. IEEE, 2009.

[93] Woo, Dong Hyuk und H.-H.S. Lee: *Extending Amdahl's Law for Energy-Efficient Computing in the Many-Core Era.* IEEE computer, 41(12):24 –31, 2008.

[94] XILINX: *Virtex-6 Family Overview*, 2010.

[95] XILINX: *Virtex-6 FPGA System Monitor*, 2010.